19. (Original) A computer program product as defined in claim 17 wherein the version-specific information relates to third party application information.

a'

20. (Original) A computer program product as defined in claim 19 wherein the third party application performs virus scanning functions and wherein the predetermined access attempt relates to a modification of the file.

21. (Original) A computer program product as defined in claim 20 wherein the version-specific attribute remains following one of the following access attempts: copy, rename or backup.

---

**REMARKS**

This Amendment is intended to fully respond to the first Office Action dated October 23, 2002. In the first Office Action, claims 1-21 were examined; claims 13-16 were objected to because of informalities; and claims 1-21 were rejected under 35 U.S.C. §102(e) as anticipated by United States Patent Number 6,366,930 to Parker et al.

Reconsideration of these objections and rejections, as they might apply to the original and amended claims in view of these remarks, is respectfully requested. Claims 1-21 are currently pending.

**Specification**

In the Office Action, the Examiner noted that the publication "Inside Windows NT, Second Edition" by David Solomon, Microsoft Press 1998 was not considered as it was not readily accessible and that a copy of the noted excerpt should be supplied. A copy of such excerpt is supplied herewith.

**Claim Objections**

Claims 13-16 were objected to because of the perceived informality of the term "may be" and whether the Applicant intended to claim the recited limitation. Further, claims 15-16 were objected to because of the use of the term "system" instead of the term "computer-readable

medium." Claims 13, 15 and 16 have been amended to resolve these issues. Claim 1 has also been amended to improve its form.

**Claim Rejections – 35 U.S.C. §102(e) (Claims 1-21)**

Regarding the prior art rejections, the Applicant respectfully traverses the 102 rejections because the Examiner has failed to substantiate a prima facie case of anticipation because at least one of the requirements of a prima facie case is absent. That is, the cited reference does not identically disclose all the limitations of the independent claims, as required by 35 U.S.C. §102.

The Office Action mailed on October 23, 2002 cites U.S. Patent No. 6,366,930 to Parker et al (hereinafter "Parker") as a 102 reference against claims 1-21. Each independent claim of the present application recites the use of a "version-specific attribute." The version-specific attribute maintains version information **related to the application that created the attribute or some other application**, relatively independent of the file itself. (See the Specification at Page 16, Lines 13 - 18) Consequently, in a particular example, future versions of a virus-scanning or replicating application are able to recognize whether existing files have been scanned or replicated **with the most recent version of the scanning or replicating application**. The version-specific attribute of the present invention does not necessarily maintain version information for the particular file and/or the particular application that created the file.

Parker does not show or describe such a version-specific attribute, as defined in the present application. Parker relates to generating unique signatures for each file based on the file itself, including the file data and/or the existing file attributes. These unique signatures are then used to determine whether a file is a new file or whether the file has been deleted or modified. Clearly the unique signatures described in Parker, since they are calculated using file dependent data, may relay information regarding the version of the file itself. This should not be confused with the present invention, however, and its version information *related to the application that created the attribute*. Therefore, Parker does not disclose version-specific attributes as defined in the present application.

Furthermore, Parker does not mention the use of the version information related to the application that created the attribute as a means of determining whether a file has been scanned

or replicated with the most recent version. Consequently, there is no suggestion of creating such an attribute to store or maintain such information. Indeed, with respect to the virus-scanning scenario, Parker uses the fact that a file may have been modified to indicate that a virus scan **should be** performed, by yet another virus-scanning application. "If any executable files have changed, the condition is identified as file corruption and a possible virus situation." Parker, Col. 9, Lines 35-37. On the other hand, using the present invention, the virus-scanning application typically scans a file unless the proper version-specific attribute is present, i.e., the virus-scanning application determines that a scan **should not be** performed.

Under 35 U.S.C. § 102, a reference must show or describe each and every element claimed in order to anticipate the claims. *Verdegaal Bros. v. Union Oil Co. of California* 814 F.2d 628 (Fed. Cir. 1987) ("A claim is anticipated only if each and every element as set forth in the claim is found, either expressly or inherently described, in a single prior art reference.") Parker does not expressly or inherently describe version-specific information related to the version of the application that created the attribute. Moreover, Parker does not show the use of such version-specific information to determine whether further actions should be performed, e.g., virus scanning or replicating. Consequently, the Parker reference, as a matter of law, cannot anticipate the independent claims of the present application. For at least this reason, the claims of the present application are believed to be allowable.

Furthermore, claims 4 and 14 recite additional limitations related to the version-specific attribute, i.e., the meta information and the mask information, wherein the mask information provides information related to which events, e.g., modification of a file, etc., cause the version-specific attribute to be invalidated. Additionally, claims 15 and 20 further provide that the version specific information store information related to a virus definition file. Such limitations are not shown or described in the Parker reference either expressly or inherently, such that these other claims are believed to be allowable over Parker for at least these additional reasons.

## Conclusion

As originally filed, the present application included 21 claims, 4 of which were independent. As amended, the present application includes 21 claims, 4 of which are

independent. Accordingly, it is believed that no further fees are due with this Amendment and Response. However, the Commissioner is hereby authorized to charge any deficiencies or credit any overpayment with respect to this patent application to deposit account number 13-2725.

In light of the above remarks and amendments, it is believed that the application is now in condition for allowance, and such action is respectfully requested. Should any additional issues need to be resolved, the Examiner is requested to telephone the undersigned to attempt to resolve those issues.

Respectfully submitted,

Dated: 1/22/03

27488
PATENT TRADEMARK OFFICE

Timothy B. Scull, #42,137
MERCHANT & GOULD P.C.
P.O. Box 2903
Minneapolis, MN 55402-0903
303.357.1648

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

| | | | |
|---|---|---|---|
| Applicant: | Jonathan S. Goldick | Examiner: | Nguyen, Cindy |
| Serial No.: | 09/750366 | Group Art Unit: | 2171 |
| Filed: | December 27, 2000 | Docket No.: | MS154771.1 / MG40062.0095US01 |
| Title: | METHOD AND APPARATUS FOR VERSION-SPECIFIC PROPERTIES IN FILE | | |



**CLAIM AMENDMENT - MARKUP**

Please amend claims 13, 15 and 16 as shown below. All claims are provided for convenience.

1. (Amended) A method of providing version-specific information associated with a file stored in a computer system to an application, the method comprising:

receiving a request to create a version-specific attribute, wherein the attribute is associated with the file;

maintaining the version-specific attribute to reflect relevant updates to the file by automatically invalidating the version-specific information in response to a predetermined event [access requests];

receiving a request by the application to evaluate the version-specific attribute; and

providing the version specific information to the application in response to the request to evaluate the version specific attribute.

13. (Amended) A computer-readable medium having stored thereon a data structure, wherein the data structure comprises:

an actual file data section containing actual file data;

a header section; and

a version specific attribute section, wherein the version-specific attribute section created by a third party application, and wherein the version-specific attribute is [may be] invalidated in response to a predetermined event.

15. (Amended) A [system] computer readable medium as defined in claim 14 wherein the third party application performs virus scanning functions and wherein the version information section stores information related to a virus definition file.

16. (Amended) A [system] computer readable medium as defined in claim 14 wherein the predetermined event relates to a modification of the data structure.

CHAPTER NINE

# Windows NT File System (NTFS)

This chapter details the internal structure and operation of the Microsoft Windows NT File System (hereafter referred to as NTFS). After reviewing the NTFS design goals and major features, I'll describe the NTFS on-disk structure and explain how NTFS implements transaction-based file system recovery. Finally, I'll cover the implementation of the optional fault tolerance support.

Windows NT 5.0 **NOTE** Some of the exciting NTFS extensions being introduced in Windows NT 5.0 are noted throughout this chapter. For a complete list of these new extensions, see Chapter 10.

## NTFS Design Goals and Features

In 1988, Microsoft already supported two file systems—the FAT file system for MS-DOS and Microsoft Windows and the high-performance file system (HPFS) for OS/2. Unfortunately, both of these file systems suffered from limitations that made them either less reliable than a file system for Windows NT should be or unable to handle the large system configurations that were expected to run the Windows NT operating system. After careful consideration, the Windows NT team decided to create a new file system—NTFS. Although the design for NTFS was new, it was influenced by FAT and HPFS as well as by certain features required by the POSIX standard.

The following section describes the requirements that drove the design of NTFS. The subsequent section examines the advanced features of NTFS.

### High-End File System Requirements

MS-DOS uses the FAT file system, which was originally designed for floppy disks of a relatively small size, generally 1 MB or less. As hard disks became the standard storage device for personal computers and over time grew larger, they began to stretch the limits of the FAT file system. The OS/2 operating system introduced HPFS to address some of the limitations of the FAT file system. For

example, HPFS greatly improved file access times for large directories and could be used on hard disks up to 4 GB in size. HPFS was later expanded to support disk sizes up to 2 TB (terabytes), or approximately 2 trillion bytes.

The FAT file system worked well for small disks, and HPFS added some new capabilities, greater file access efficiency, and support for larger media. However, neither file system was suitable for mission-critical applications that required recoverability, security, data redundancy and fault tolerance, and support for even larger storage media than HPFS provided.

### Recoverability

As far as disk I/O is concerned, personal computer users have tended to care most about speed—above all, they've usually just wanted to get their work done fast. As Windows NT moves the personal computer into more businesses and corporations, however, the reliability of the data stored on the system becomes increasingly important relative to the speed with which a user can access data on a disk drive. In other words, if the system fails and a disk drive is corrupted or becomes inaccessible, the speed of the preceding I/O operations is largely irrelevant.

To address the requirement for reliable data storage and data access, NTFS provides file system recovery based on a transaction-processing model. *Transaction processing* is a technique for handling modifications to a database so that system failures don't affect the correctness or integrity of the database. The basic tenet of transaction processing is that some database operations, called transactions, are all-or-nothing propositions. (A *transaction* is defined as an I/O operation that alters file system data or changes the volume's directory structure.) The separate disk updates that make up the transaction must be executed *atomically*; that is, once the transaction begins to execute, all of its disk updates must be completed. If a system failure interrupts the transaction, the part that has been completed must be undone, or *rolled back*. The rollback operation returns the database to a previously known and consistent state, as if the transaction had never occurred.

NTFS uses the transaction-processing model to implement its file system recovery feature. If a program initiates an I/O operation that alters the structure of the NTFS—that is, changes the directory structure, extends a file, allocates space for a new file, and so on—NTFS treats that operation as an atomic transaction. It guarantees that the transaction is either completed or, if the system fails while executing the transaction, rolled back. The details of how NTFS does this is explained in the section "Recoverability Support" on page 426.

In addition, NTFS uses redundant storage for vital file system information so that if one location on the disk goes bad, NTFS can still access the volume's critical file system data. This redundancy of file system data contrasts

with the on-disk structures of both the FAT file system and HPFS, which have single sectors containing critical file system data. If a read error occurs in one of these sectors, an entire volume is lost.

### Security

Data security is crucial to customers who process private or sensitive information—banks, hospitals, and national defense-related agencies, for example. Such customers need guarantees that their data will be secure from unauthorized access.

Security in NTFS is derived directly from the Windows NT object model. (For more information on NTFS security, see Chapter 6.) An open file is implemented as a file object with a security descriptor stored on disk as a part of the file. Before a process can open a handle to any object, including a file object, the Windows NT security system verifies that the process has appropriate authorization to do so. The security descriptor, combined with the requirement that a user log on to the system and provide an identifying password, ensures that no process can access a file unless given specific permission to do so by a system administrator or by the file's owner. (For more information about security descriptors, see the section on page 310 in Chapter 6, and for more details about file objects, see the section on page 341 in Chapter 7.)

### Data Redundancy and Fault Tolerance

In addition to recoverability of file system data, some customers require that their own data not be endangered by a power outage or catastrophic system failure. The NTFS recovery capabilities do ensure that the file system on a volume remains accessible, but they make no guarantees for complete recovery of user files. For applications that can't risk losing file data, data redundancy provides an extra level of protection.

The Windows NT layered driver model (explained in Chapter 7) is used to provide fault tolerant disk support. NTFS communicates with a fault tolerant disk driver, which in turn communicates with a hard disk driver to write data to disk. This communication allows a Windows NT system to establish fault tolerant disk storage by installing an additional driver. The fault tolerant driver can *mirror*, or duplicate, data from one disk onto another disk so that a redundant copy can always be retrieved. This support is commonly called *RAID level 1*. The fault tolerant driver also allows data to be written in *stripes* across three or more disks, using the equivalent of one disk to maintain parity information. If the data on one disk is lost or becomes inaccessible, the driver can reconstruct the disk's contents by means of exclusive-OR operations. This support is called *RAID level 5*.

## Large Disks and Large Files

Engineering and other scientific applications often store and process extremely large quantities of information. Hard disks with over 8 GB of storage and disk arrays with 100 GB to 500 GB of storage are no longer uncommon. NTFS supports very large disks and large files more efficiently than does either the FAT file system or HPFS.

Until the introduction of Microsoft Windows 95 OSR2, the FAT file system used a table 16 bits wide to record the allocation status of a disk volume. Because a volume is divided into same-sized allocation units—called *clusters*—and each cluster must be uniquely numbered using 16 bits, FAT can support a maximum of $2^{16}$, or 65,536, clusters per volume (although the FAT reserves some of this space for itself). A single FAT volume is limited to containing 65,518 files (the maximum number of available clusters), regardless of the disk size.

Windows NT 5.0 ▶ **NOTE** Windows NT 5.0 will support FAT32, the enhanced version of FAT shipped as part of Microsoft Windows 95 OEM Service Release 2 (as well as Microsoft Windows 98). FAT32 alleviates some of the restrictions of the FAT16 design by allowing smaller clusters (4 KB for drives up to 8 GB) as well as support for hard disk sizes larger than 2 GB. FAT32 also has the ability to relocate the root directory and use the backup copy of the FAT instead of the default copy. The boot record on FAT32 drives has been expanded to include a backup of critical data structures, making FAT32 partitions less susceptible to a single point of failure than are existing FAT16 volumes. Finally, the root directory on a FAT32 drive is now an ordinary cluster chain, so it can be located anywhere on the drive. For this reason, the previous limitations on the number of root directory entries no longer exist.

HPFS uses 32 bits to enumerate its allocation units, a strategy that yields $2^{32}$, or over 4 billion, numbers. HPFS uses signed numbers, however, which reduces this number to 2 billion possible allocation units on an HPFS volume. Rather than clusters, HPFS allocates disk space in terms of physical sectors, each set at 512 bytes. This lack of flexibility can be a problem, particularly in Asian markets, where disk drives commonly have a hardware sector size of 1024 bytes. HPFS can't be used on such drives because disks can't allocate space in increments smaller than their hardware sector size. HPFS is also limited to a maximum file size of 4 GB.

NTFS allocates clusters and uses 64 bits to number them, which results in a possible $2^{64}$ (over 16,000,000,000,000,000,000, or 16 billion billion) clusters, each up to 64 KB. Each file can be $2^{64}$ bytes long, which should satisfy data storage requirements for some time to come. As in the FAT file system, the

cluster size in NTFS is adjustable, but it is not required to grow proportionally to the disk size. NTFS uses a cluster size of 512 bytes on small disks and a maximum cluster size of 64 KB on large disks. Although NTFS uses a 64-bit (8-byte) disk address to represent each run (disk allocation), it "encodes" the addresses so that they occupy only 3 to 5 bytes per run. (Look ahead to Figure 9-17 to see an example of address encoding.) HPFS uses 12 bytes to represent each run.

## Additional Features in NTFS

In addition to NTFS being recoverable, secure, reliable, and efficient for mission-critical systems, it includes the following advanced features that allow it to support a broad range of applications.

### Multiple Data Streams

In NTFS, each unit of information associated with a file, including its name, its owner, its time stamps, its contents, and so on, is implemented as a file attribute (object attribute). Each attribute consists of a single *stream,* that is, a simple sequence of bytes. This generic implementation makes it easy to add more attributes (and therefore more streams) to a file. Because a file's data is "just another attribute" of the file and because new attributes can be added, NTFS files (and file directories) can contain multiple data streams.

An NTFS file has one default data stream, which has no name. An application can create additional, named data streams and access them by referring to their names. To avoid altering the Microsoft Win32 I/O APIs, which take a string as a filename argument, the name of the data stream is specified by appending a colon (:) to the filename. Because the colon is a reserved character, it can serve as a separator between the filename and the data stream name, as illustrated in this example:

```
myfile.dat:stream2
```

Each stream has a separate allocation size (how much disk space has been reserved for it), an actual size (how many bytes the caller has used), and a valid data length (how much of the stream has been initialized). In addition, each stream is given a separate file lock that is used to lock byte ranges and to allow concurrent access. To reduce processing overhead, sharing is done per file rather than per stream.

The one component in Windows NT that uses multiple data streams is the Apple Macintosh file server support that comes with Windows NT Server. Macintosh systems use two streams per file— one to store data and the other to store resource information, such as the file type and the icon used to represent the file. Because NTFS allows multiple data streams, a Macintosh user can copy an entire Macintosh folder (analogous to a directory) to a Windows NT Server,

and another Macintosh user can copy the folder from the server without losing resource information. Other applications can use the multiple data stream feature as well. A backup utility, for example, might use an extra data stream to store backup-specific time stamps on files. Or an archival utility might implement hierarchical storage in which files that are older than a certain date or that haven't been accessed for a specified period of time are moved to tape. The utility could copy the file to tape, set the file's default data stream to 0, and add a data stream that specifies the name and location of the tape on which the file is stored.

### Unicode-Based Names

Like Windows NT as a whole, NTFS is fully Unicode enabled, using Unicode characters to store names of files, directories, and volumes. Unicode, a 16-bit character-coding scheme, allows each character in each of the world's major languages to be uniquely represented, which aids in moving data easily from one country to another. Unicode is an improvement over traditional representation of international characters—using a double-byte coding scheme that stores some characters in 8 bits and others in 16 bits, a technique that requires loading various code pages to establish the available characters. Because Unicode has a unique representation for each character, it doesn't depend on which code page is loaded. Each directory and filename in a path name can be as many as 255 characters long and can contain Unicode characters, embedded spaces, and multiple periods.

### General Indexing Facility

The NTFS architecture is structured to allow indexing of file attributes on a disk volume. This structure enables the file system to efficiently locate files that match certain criteria—for example, all the files in a particular directory. The FAT file system indexes filenames but doesn't sort them, making lookups in large directories slow. HPFS indexes and sorts filenames as NTFS does, but the design of NTFS allows for indexing other file attributes as well.

Windows NT 5.0 **NOTE** In Windows NT 5.0, NTFS is being extended to index other attributes, such as object IDs (enterprise-wide unique identifiers for files). For more information on the additional indexing capabilities planned for the NTFS in Windows NT 5.0, see Chapter 10.

### Dynamic Bad-Cluster Remapping

Ordinarily, if a program tries to read data from a bad-disk sector, the read operation fails and the data in the allocated cluster becomes inaccessible. If the disk is formatted as a fault tolerant NTFS volume, however, the Windows NT fault tolerant driver dynamically retrieves a good copy of the data that was

stored on the bad sector and then sends NTFS a warning that the sector is bad. NTFS allocates a new cluster, replacing the cluster in which the bad sector resides, and copies the data to the new cluster. It flags the bad cluster and no longer uses it. This data recovery and dynamic bad-cluster remapping is an especially useful feature for file servers and fault tolerant systems or for any application that can't afford to lose data. If the fault tolerant driver isn't loaded when a sector goes bad, NTFS still replaces the cluster and doesn't reuse it, but it can't recover the data that was on the bad sector.

### POSIX Support

As explained in Chapter 1, one of the mandates for Windows NT was to fully support the POSIX.1 standard. In the file system area, the POSIX standard requires support for case-sensitive file and directory names, a "file-change-time" time stamp (which is different than the MS-DOS "time-last-modified" stamp), and hard links (multiple directory entries that point to the same file). NTFS implements each of these features.

### EXPERIMENT: Creating a Hard Link

To create a hard link, use the ln utility in the Windows NT Resource Kit. For example, type the following lines of code:

```
C:\>cd \ntreskit\posix
C:\NTRESKIT\POSIX>dir ln.exe
 Volume in drive C has no label.
 Volume Serial Number is 28BC-F247
 Directory of C:\NTRESKIT\POSIX
03/01/98  12:00a              90,960 LN.EXE
               1 File(s)      90,960 bytes
                         317,863,936 bytes free

C:\NTRESKIT\POSIX>ln ln.exe foo.exe
ln: ln.exe: No such file or directory

C:\NTRESKIT\POSIX>ln LN.EXE foo.exe
```

The second to last command failed because the POSIX subsystem processes filenames as case-sensitive (Win32 does not). So in the last command, referencing LN.EXE (uppercase letters) worked and created the hard link—a directory entry called "foo.exe" was created to point to the same file that LN.EXE points to. If you delete LN.EXE, the file remains because there is still a link to it (foo.exe). When the last link to a file is deleted, the space for the file itself is released.

# NTFS Internal Structure

As described in Chapter 7, in the framework of the Windows NT I/O system, NTFS and other file systems are loadable device drivers that run in kernel mode. They are invoked indirectly by applications that use Win32 or other I/O APIs (such as POSIX). As Figure 9-1 shows, the Windows NT environment subsystems call Windows NT system services, which in turn locate the appropriate loaded drivers and call them. (For a description of system service dispatching, see page 99 in Chapter 3.)

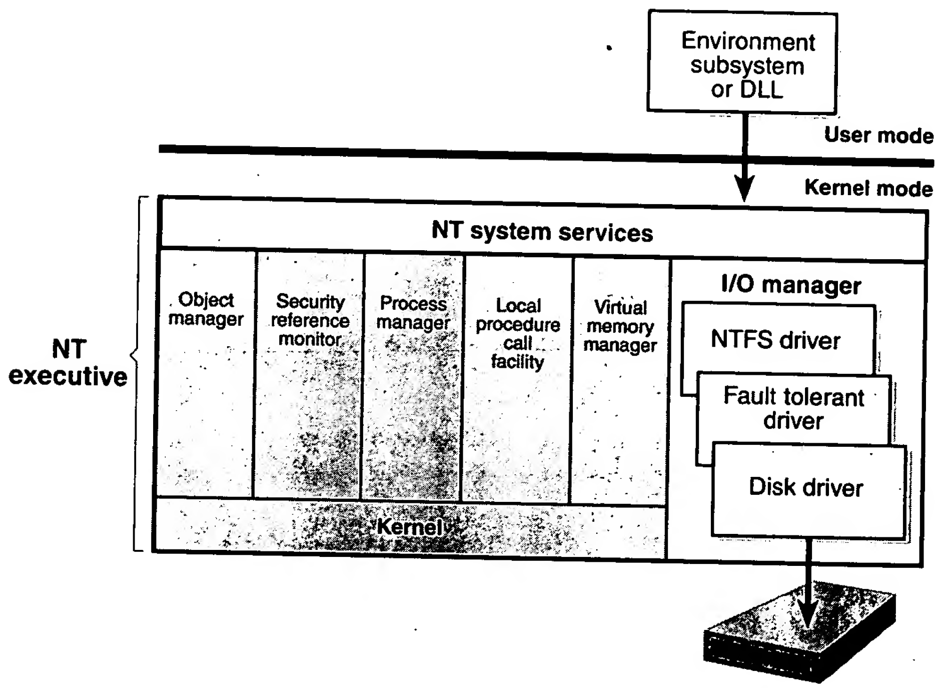


**Figure 9-1**
*Components of the Windows NT I/O system*

The layered drivers pass I/O requests to one another by calling the Windows NT executive's I/O manager. Relying on the I/O manager as an intermediary allows each driver to maintain independence so that it can be loaded or unloaded without affecting other drivers. In addition, the NTFS driver interacts with the three other NT executive components, shown in the left side of Figure 9-2, that are closely related to file systems.

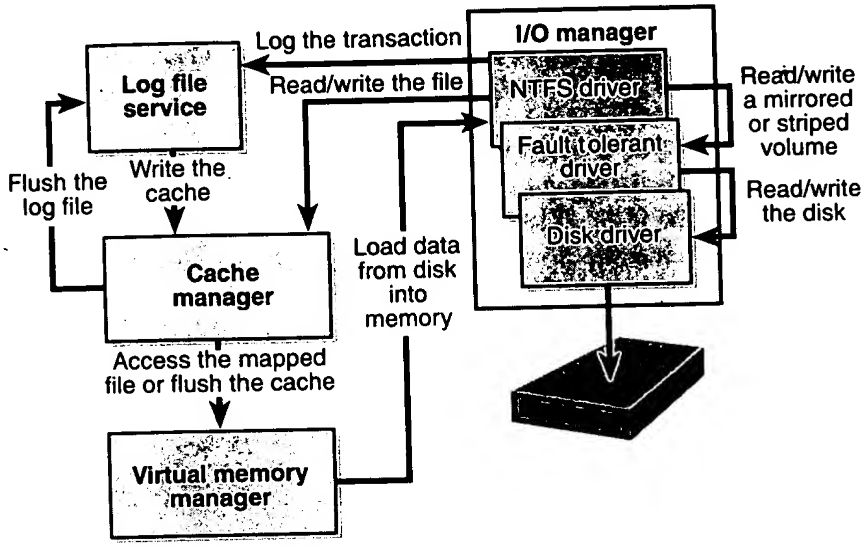


**Figure 9-2**
*NTFS and related components*

The *log file service* (*LFS*) is the part of NTFS that provides services for maintaining a log of disk writes. The log file it writes is used to recover an NTFS-formatted volume in the case of a system failure.

The *cache manager* is the component of the Windows NT executive that provides systemwide caching services for NTFS and other file system drivers, including network file system drivers (servers and redirectors). All file systems implemented for Windows NT access cached files by mapping them into virtual memory and then accessing the virtual memory. The cache manager provides a specialized file system interface to the Windows NT *virtual memory manager* for this purpose. When a program tries to access a part of a file that is not loaded into the cache (a *cache miss*), the memory manager calls NTFS to access the disk driver and obtain the file contents from disk. The cache manager optimizes disk I/O by using its *lazy writer*, a set of system threads that call the memory manager to flush cache contents to disk as a background activity (asynchronous disk writing). (For a complete description of the cache manager, see Chapter 8.)

NTFS participates in the Windows NT object model by implementing files as objects. This implementation allows files to be shared and protected by the object manager, the component of Windows NT that manages all executive-level objects. (The object manager is described on page 101 in Chapter 3.)

An application creates or accesses a file just as it does other Windows NT objects: by means of object handles. By the time an I/O request reaches NTFS, the Windows NT object manager and security system have already verified that the calling process has the authority to access the file object in the way it is attempting to. The security system has compared the caller's access token to the entries in the access control list for the file object. (See Chapter 6 for more information about access control lists.) The I/O manager has also transformed the file handle into a pointer to a file object. NTFS uses the information in the file object to access the file on disk.

Figure 9-3 shows the data structures that link a file handle to the file system's on-disk structure.

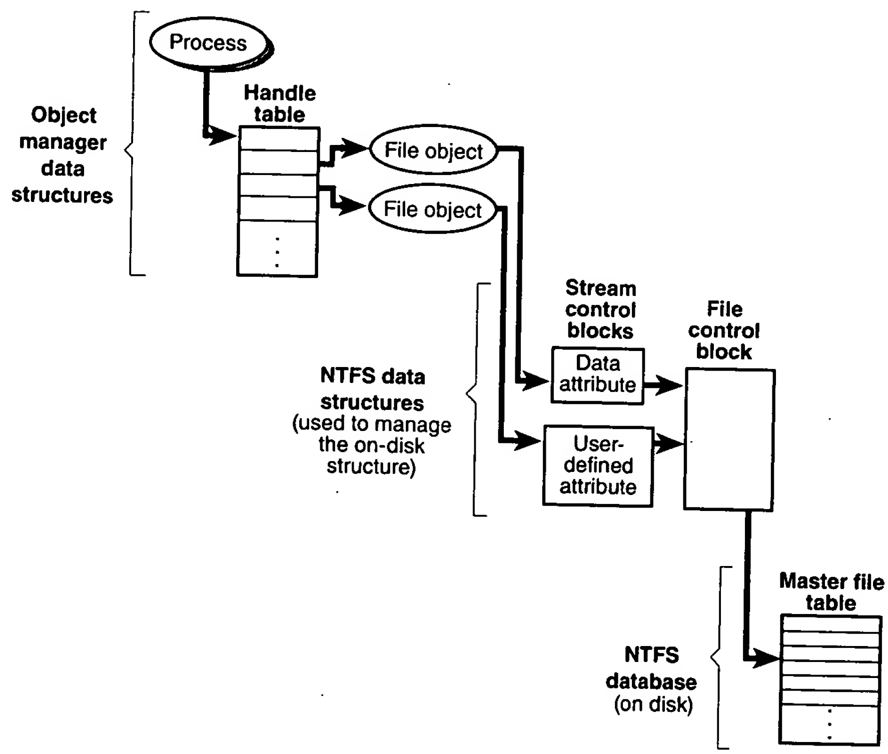


**Figure 9-3**
*NTFS data structures*

By the time the I/O system calls NTFS, the handle has been translated to a pointer to a file object. NTFS then follows several pointers to get from the file object to the location of the file on disk. As Figure 9-3 shows, a file object, which represents a single call to the open-file system service, points to a *stream*

*control block* (*SCB*) for the file attribute that the caller is trying to read or write. In Figure 9-3, a process has opened both the data attribute and a user-defined attribute for the file. The SCBs represent individual file attributes and contain information about how to find specific attributes within a file. All the SCBs for a file point to a common data structure called a *file control block* (*FCB*). The FCB contains a pointer (actually, a file reference, explained in the section "File Reference Numbers" later in this chapter) to the file's record in the disk-based master file table (or MFT), which is described in detail in the following section.

# NTFS On-Disk Structure

This section describes the on-disk structure of an NTFS volume, including how disk space is divided and organized into clusters, how files are organized into directories, how the actual file data and attribute information is stored on disk, and finally, how NTFS data compression works.

## Volumes

The structure of NTFS begins with a volume. A *volume* corresponds to a logical partition on a disk, and it is created when you format a disk or part of a disk for NTFS. You can also create a fault tolerant volume that spans multiple disks by using the Windows NT Disk Administrator utility.

A disk can have one volume or several. NTFS handles each volume independently of the others. Three sample disk configurations for a 150-MB hard disk are illustrated in Figure 9-4.

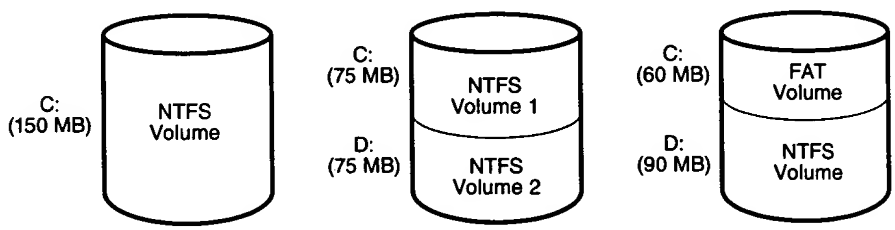


**Figure 9-4**
*Sample disk configurations*

A volume consists of a series of files plus any additional unallocated space remaining on the disk partition. In the FAT file system, a volume also contains areas specially formatted for use by the file system. An NTFS volume, however, stores all file system data, such as bitmaps and directories, and even the system bootstrap, as ordinary files.

## Clusters

NTFS is like the FAT file system in that it uses the cluster as its fundamental unit of disk allocation. The cluster size on a volume, or the *cluster factor,* is established when a user formats the volume with either the Format command or the Disk Administrator utility. The cluster factor varies with the size of the volume, but it is an integral number of physical sectors, always a power of 2 (1 sector, 2 sectors, 4 sectors, 8 sectors, and so on), as shown in Figure 9-5. The cluster factor is expressed as the number of bytes in the cluster, such as 512 bytes, 1 KB, or 2 KB.

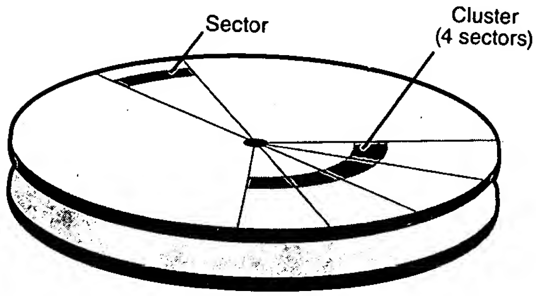


**Figure 9-5**
*Sectors and a cluster on a disk*

Internally, NTFS refers only to clusters and is unaware of a disk's sector size. NTFS uses the cluster as its unit of allocation in order to maintain its independence from physical sector sizes. This independence allows NTFS to efficiently support very large disks by using a larger cluster size or to support nonstandard disks that have a sector size other than 512 bytes. On a larger volume, use of a larger cluster size can reduce fragmentation and speed allocation, at a small cost in terms of wasted disk space. The Format command (available from the Windows NT command prompt) as well as the Format command on the Tools menu in Disk Administrator choose a default cluster size based on the volume size, but you can override this size.

**NOTE** The default cluster size for small disks (up to 512 MB) is 512 bytes (or the hardware sector size if it is larger than 512 bytes). For disks up to 1 GB, the default cluster size is 1 KB. For disks between 1 GB and 2 GB, the default cluster size is 2 KB. For disks larger than 2 GB, the default cluster size is 4 KB. This default balances the inherent trade-off between the disk fragmentation that can occur with too small a cluster size and the wasted space (internal fragmentation) that can occur with too large a cluster size.

NTFS refers to physical locations on a disk by means of *logical cluster numbers* (*LCNs*). LCNs are simply the numbering of all clusters from the beginning of the volume to the end. To convert an LCN to a physical disk address, NTFS multiplies the LCN by the cluster factor to get the physical byte offset on the volume, as the disk driver interface requires. NTFS refers to the data within a file by means of *virtual cluster numbers* (*VCNs*). VCNs number the clusters belonging to a particular file from 0 through *m*. VCNs are not necessarily physically contiguous, however; they can be mapped to any number of LCNs on the volume.

## Master File Table (MFT)

In NTFS, all data stored on a volume is contained in a file, including the data structures used to locate and retrieve files, the bootstrap data, and the bitmap that records the allocation state of the entire volume (the NTFS metadata). Storing everything in files allows the file system to easily locate and maintain the data, and each separate file can be protected by a security descriptor. In addition, if a particular part of the disk goes bad, NTFS can relocate the metadata files to prevent the disk from becoming inaccessible.

The *master file table* (MFT) is the heart of the NTFS volume structure. The MFT is implemented as an array of file records. The size of each file record is fixed at 1 KB, regardless of cluster size. (The structure of a file record is described in the "File Records" section on page 410.) Logically, the MFT contains one row for each file on the volume, including a row for the MFT itself. In addition to the MFT, each NTFS volume includes a set of *metadata files* containing the information that is used to implement the file system structure. Each of these NTFS metadata files has a name that begins with a dollar sign ($), although the signs are hidden. For example, the filename of the MFT is $MFT. The rest of the files on an NTFS volume are normal user files and directories, as shown in Figure 9-6 on the following page.

Usually, each MFT record corresponds to a different file. If a file has a large number of attributes or becomes highly fragmented, however, more than one file record might be needed. In such cases, the first record, which stores the locations of the others, is called the *base file record*.

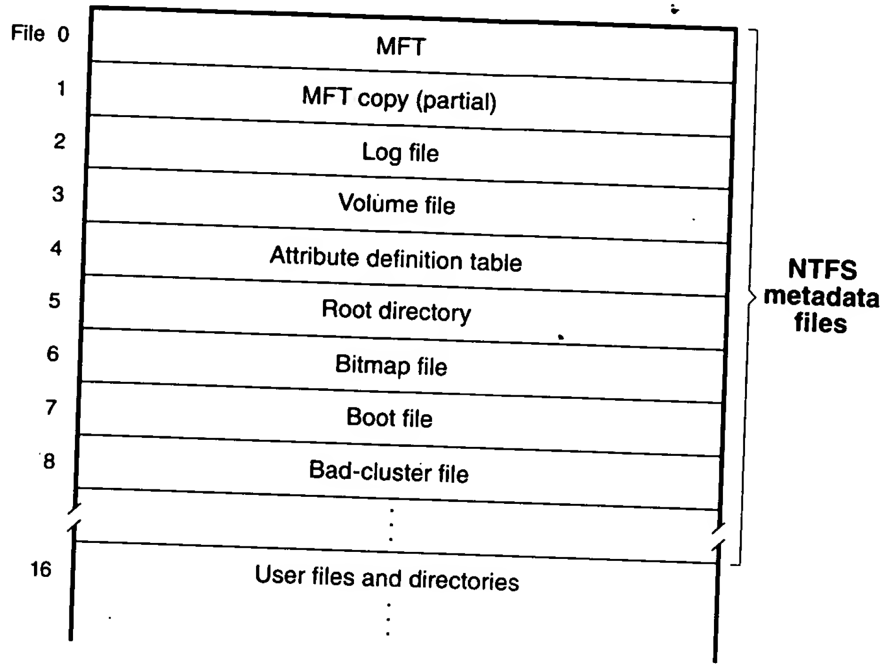


**Figure 9-6**
*File records for NTFS metadata files in the MFT*

When it first accesses a volume, NTFS must *mount* it—that is, prepare it for use. To mount the volume, NTFS looks in the boot file (defined on page 409) to find the physical disk address of the MFT. The MFT's own file record is the first entry in the table; the second file record points to a file located in the middle of the disk called the *MFT mirror* (filename $MFTMirr) that contains a copy of the first few rows of the MFT. This partial copy of the MFT is used to locate metadata files if part of the MFT file can't be read for some reason.

Once NTFS finds the file record for the MFT, it obtains the VCN-to-LCN mapping information in the file record's data attribute, decompresses it, and stores it in memory. This mapping information tells NTFS where the runs composing the MFT are located on the disk. NTFS then decompresses the MFT records for several more metadata files and opens the files. Next, NTFS performs its file system recovery operation (described in the section "Recovery" on page 436), and finally, it opens its remaining metadata files. The volume is now ready for user access.

As the system runs, NTFS writes to another important metadata file, the *log file* (filename $LogFile). NTFS uses the log file to record all operations that affect the NTFS volume structure, including file creation or any commands, such as Copy, that alter the directory structure. The log file is used to recover an NTFS volume after a system failure.

Another entry in the MFT is reserved for the root directory (also known as "\"). Its file record contains an index of the files and directories stored in the root of the NTFS directory structure. When NTFS is first asked to open a file, it begins its search for the file in the root directory's file record. After opening a file, NTFS stores the file's MFT file reference so that it can directly access the file's MFT record when it reads and writes the file later.

NTFS records the allocation state of the volume in the *bitmap file* (filename $Bitmap). The data attribute for the bitmap file contains a bitmap, each of whose bits represents a cluster on the volume, identifying whether the cluster is free or has been allocated to a file.

Another important system file, the *boot file* (filename $Boot), stores the Windows NT bootstrap code. For the system to boot, the bootstrap code must be located at a specific disk address. During formatting, however, the Format utility defines this area as a file by creating a file record for it. Creating the boot file allows NTFS to adhere to its rule of making everything on the disk a file. The boot file as well as NTFS metadata files can be individually protected by means of the security descriptors that are applied to all Windows NT objects. Using this "everything on the disk is a file" model also means that the bootstrap can be modified by normal file I/O, although the boot file is currently protected from editing.

## EXPERIMENT: Viewing NTFS Metadata Files

Since the NTFS metadata files are regular NTFS files, they can be seen with the directory command if you use the /a:h (hidden) qualifier and type the correct name of the file, as shown here:

```
C:\>dir/a:h $mft
 Volume in drive C has no label.
 Volume Serial Number is 28BC-F247

 Directory of C:\

03/06/97  06:11p           1,035,264 $MFT
```

NTFS also maintains a *bad-cluster file* (filename $BadClus) for recording any bad spots on the disk volume and a file known as the *volume file* (filename $Volume), which contains the volume name, the version of NTFS for which the volume is formatted, and a bit that when set signifies that a disk corruption has occurred and must be repaired by the Chkdsk utility. (The Chkdsk utility is covered in more detail later in the chapter.) Finally, NTFS maintains a file containing an *attribute definition table* (filename $AttrDef) that defines the attribute types supported on the volume and indicates whether they can be indexed, recovered during a system recovery operation, and so on.

## File Reference Numbers

A file on an NTFS volume is identified by a 64-bit value called a *file reference.* The file reference consists of a file number and a sequence number. The file number corresponds to the position of the file's file record in the MFT minus 1 (or to the position of the base file record minus 1 if the file has more than 1 file record). The file reference sequence number, which is incremented each time an MFT file record position is reused, enables NTFS to perform internal consistency checks. A file reference is illustrated in Figure 9-7.

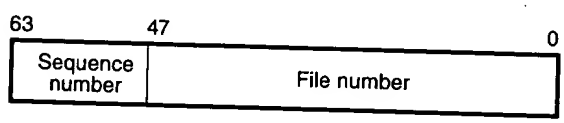


**Figure 9-7**
*File reference*

## Files Records

Instead of viewing a file as just a repository for textual or binary data, NTFS stores files as a collection of attribute/value pairs, one of which is the data it contains (called the *unnamed data attribute*). Other attributes that comprise a file include the filename, time stamp information, security descriptor, and possibly additional named data attributes. Figure 9-8 illustrates an MFT record for a small file.

Each file attribute is stored as a separate stream of bytes within a file. Strictly speaking, NTFS doesn't read and write files—it reads and writes attribute streams. NTFS supplies these attribute operations: create, delete, read (byte range), and write (byte range). The read and write services normally operate on the file's unnamed data attribute. However, a caller can specify a different data attribute by using the named data stream syntax.

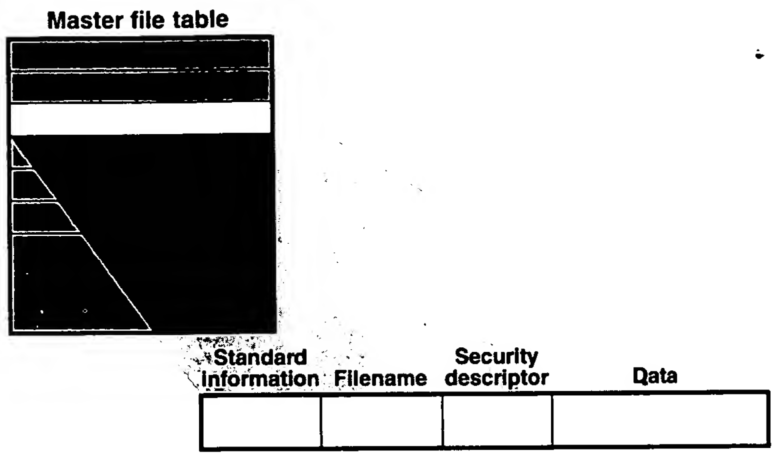


**Figure 9-8**
*MFT record for a small file*

Table 9-1 lists the standard attributes for files on an NTFS volume. (Not all attributes are present for every file.)

**Table 9-1 Standard Attributes for NTFS Files**

| Attribute | Description |
|---|---|
| Standard information | File attributes such as read-only, archive, and so on; time stamps, including when the file was created or last modified; and how many directories point to the file (its *hard link count*). |
| Filename | The file's name in Unicode characters. A file can have multiple filename attributes, as it does when a POSIX hard link to a file exists or when a file with a long name has an automatically generated "short name" for access by MS-DOS and 16-bit Microsoft Windows applications. |
| Security descriptor | The security data structure that protects the file from unauthorized access. The security descriptor attribute specifies who owns the file and who can access it. |

*(continued)*

**Table 9-1** *continued*

| Attribute | Description |
|---|---|
| Data | The contents of the file. In NTFS, a file has one default unnamed data attribute and can have additional named data attributes; that is, a file can have multiple data streams. A directory has no default data attribute but can have optional named data attributes. |
| Index root, index | Three attributes used to implement filename allocation, bitmap indexes for large directories (directories only). |
| Attribute list | A list of the attributes that make up the file and the file reference of the MFT file record in which each attribute is located. This seldom-used attribute is present when a file requires more than one MFT file record. |

Windows NT 5.0 **NOTE** To save disk space, security descriptors in Windows NT 5.0 are stored in a central file and referenced by each file record.

Each attribute in a file record has a name (optional) and a value. NTFS identifies an attribute by its name in uppercase letters preceded by a dollar sign ($), such as $FILENAME or $DATA. An attribute's value is the byte stream composing the attribute. For example, the value of the $FILENAME attribute is the file's name; the value of the $DATA attribute is whatever bytes the user stored in the file. These attribute names, however, actually correspond to numeric type codes, which NTFS uses to order the attributes within a file record. The file attributes in an MFT record are ordered by these type codes (numerically in ascending order), with some attribute types appearing more than once—if a file has multiple data attributes, for example, or multiple filenames.

## Filenames

Both NTFS and FAT allow each filename in a path to be as many as 255 characters long. Filenames can contain Unicode characters as well as multiple periods and embedded spaces. However, the FAT file system supplied with MS-DOS is limited to 8 (non-Unicode) characters for its filenames, followed by a period and a 3-character extension. Figure 9-9 provides a visual representation of the different *file namespaces* Windows NT supports and shows how they intersect.

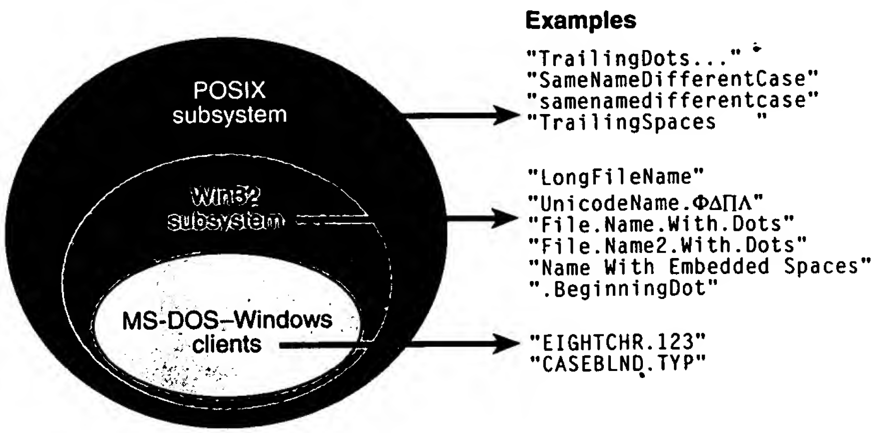


**Figure 9-9**
*Windows NT file namespaces*

The POSIX subsystem requires the biggest namespace of all the application execution environments that Windows NT supports, and therefore the NTFS namespace is equivalent to the POSIX namespace. The POSIX subsystem can create names that are not visible to Win32 and MS-DOS applications, including names with trailing periods and trailing spaces. Ordinarily, creating a file using the large POSIX namespace is not a problem because you would do that only if you intended that file to be used by the POSIX subsystem or by POSIX client systems.

The relationship between 32-bit Windows (Win32) applications and MS-DOS–Windows applications is a much closer one, however. The Win32 area in Figure 9-9 represents filenames that the Win32 subsystem can create on an NTFS volume but that MS-DOS and 16-bit Windows applications can't see. This group includes filenames longer than the 8.3 format of MS-DOS names, those containing Unicode (international) characters, those with multiple period characters or a beginning period, and those with embedded spaces. When a file is created with such a name, NTFS automatically generates an alternate, MS-DOS-style filename for the file. Windows NT displays these short names when you use the */x* option with the Dir command.

The MS-DOS filenames are fully functional aliases for the NTFS files and are stored in the same directory as the long filenames. The MFT record for a file with an autogenerated MS-DOS filename is shown in Figure 9-10.

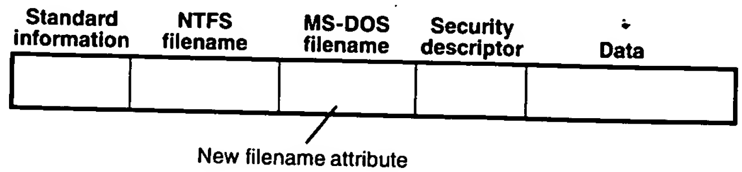


**Figure 9-10**
*MFT file record with an MS-DOS filename attribute*

The NTFS name and the generated MS-DOS name are stored in the same file record and therefore refer to the same file. The MS-DOS name can be used to open, read from, write to, or copy the file. If a user renames the file using either the long filename or the short filename, the new name replaces both of the existing names. If the new name is not a valid MS-DOS name, NTFS generates another MS-DOS name for the file.

**NOTE** POSIX hard links are implemented in a similar way. When a hard link to a POSIX file is created, NTFS adds another filename attribute to the file's MFT file record. The two situations differ in one regard, however. When a user deletes a POSIX file that has multiple names (hard links), the file record and the file remain in place. The file and its record are deleted only when the last filename (hard link) is deleted. If a file has both an NTFS name and an autogenerated MS-DOS name, however, a user can delete the file using either name.

Here's the algorithm NTFS currently uses to generate an MS-DOS name from a long filename:

1. Remove from the long name any characters that are illegal in MS-DOS names, including spaces and Unicode characters. Remove preceding and trailing periods. Remove all other embedded periods, except the last one.
2. Truncate the string before the period (if present) to six characters, and append the string "~1". Truncate the string after the period (if present) to three characters.
3. Put the result in uppercase letters. MS-DOS is case-insensitive, and this step guarantees that NTFS won't generate a new name that differs from the old only in case.
4. If the generated name duplicates an existing name in the directory, increment the "~1" string.

Table 9-2 shows the long Win32 filenames from Figure 9-9 and their NTFS-generated MS-DOS versions. The current algorithm and the examples in Figure 9-9 on page 413 should give you an idea of what NTFS-generated MS-DOS-style filenames look like. Application developers shouldn't depend on this algorithm, though, because it might change in the future.

**Table 9-2 NTFS-Generated Filenames**

| Win32 Long Name | NTFS-Generated Short Name |
|---|---|
| LongFileName | LONGFI~1 |
| UnicodeName.ΦΔΠΛ | UNICOD~1 |
| File.Name.With.Dots | FILENA~1.DOT |
| File.Name2.With.Dots | FILENA~2.DOT |
| Name With Embedded Spaces | NAMEWI~1 |
| .BeginningDot | BEGINN~1 |

## Resident and Nonresident Attributes

If a file is small, all its attributes and their values (its data, for example) fit in the file record. When the value of an attribute is stored directly in the MFT, the attribute is called a *resident attribute.* (In Figure 9-8, for example, all attributes are resident.)

Each attribute begins with a standard header containing information about the attribute, information that NTFS uses to manage the attributes in a generic way. The header, which is always resident, records whether the attribute's value is resident or nonresident. For resident attributes, the header also contains the offset from the header to the attribute's value and the length of the attribute's value, as Figure 9-11 on the following page illustrates for the filename attribute.

When an attribute's value is stored directly in the MFT, the time it takes NTFS to access the value is greatly reduced. Instead of looking up a file in a table and then reading a succession of allocation units to find the file's data (as the FAT file system does, for example), NTFS accesses the disk once and retrieves the data immediately.

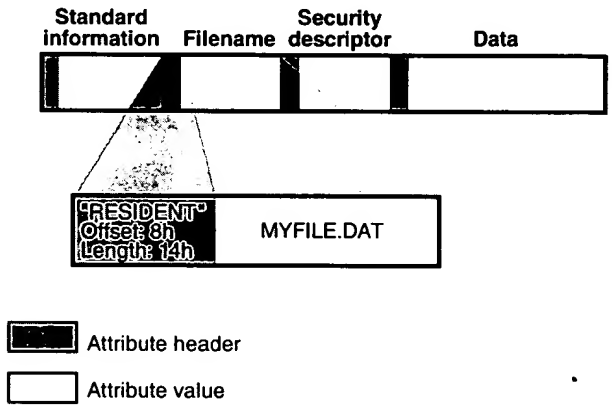


**Figure 9-11**
*Resident attribute header and value*

The attributes for a small directory, as well as for a small file, can be resident in the MFT, as Figure 9-12 shows. For a small directory, the index root attribute contains an index of file references for the files and the subdirectories in the directory.

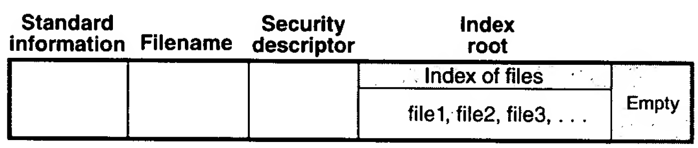


**Figure 9-12**
*MFT file record for a small directory*

Of course, many files and directories can't be squeezed into the 1-KB fixed-size MFT record. If a particular attribute, such as a file's data attribute, is too large to be contained in the MFT file record, NTFS allocates a 2-KB area on the disk (4 KB for volumes with a 4 KB or larger cluster size), separate from the MFT. This area, called a *run* (or an *extent*), stores the value of the attribute (the file's data, for example). If the attribute's value later grows (if a user appends data to the file, for instance), NTFS allocates another run for the additional data. Attributes whose values are stored in runs rather than in the MFT are called *nonresident attributes*. The file system decides whether a particular attribute is resident or nonresident; the location of the data is transparent to the process accessing it.

When an attribute is nonresident, as the data attribute for a large file might be, its header contains the information NTFS needs to locate the attribute's value on the disk. Figure 9-13 shows a nonresident data attribute stored in two runs.

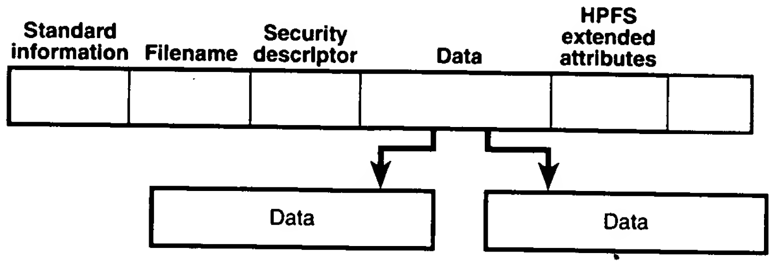


**Figure 9-13**
*MFT file record for a large file with two data runs*

Among the standard attributes, only those that can grow can be nonresident. For files, the attributes that can grow are the security descriptor, the data, and the attribute list (not shown in Figure 9-13). The standard information and filename attributes are always resident.

A large directory can also have nonresident attributes (or parts of attributes), as Figure 9-14 shows. In this example, the MFT file record doesn't have enough room to store the index of files that make up this large directory. A part of the index is stored in the index root attribute, and the rest of the index is stored in nonresident runs called *index buffers*. The index root, index allocation, and bitmap attributes are shown here in a simplified form. They are described in more detail in the next section. The standard information and filename attributes are always resident. The header and at least part of the value of the index root attribute are also resident for directories.

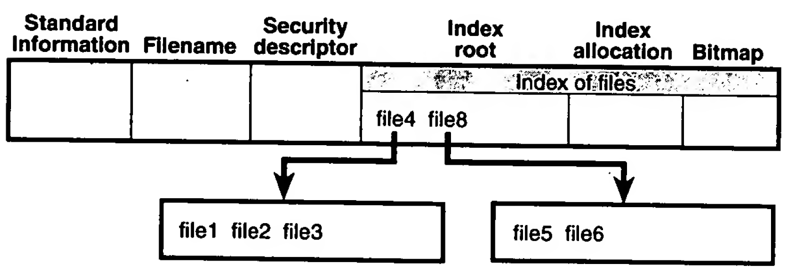


**Figure 9-14**
*MFT file record for a large directory with a nonresident filename index*

When a file's (or a directory's) attributes can't fit in an MFT file record and separate allocations are needed, NTFS keeps track of the runs by means of VCNs. LCNs represent the sequence of clusters on an entire volume from 0 through *n*. VCNs number the clusters belonging to a particular file from 0 through *m*. For example, the clusters in the runs of a nonresident data attribute are numbered as shown in Figure 9-15.

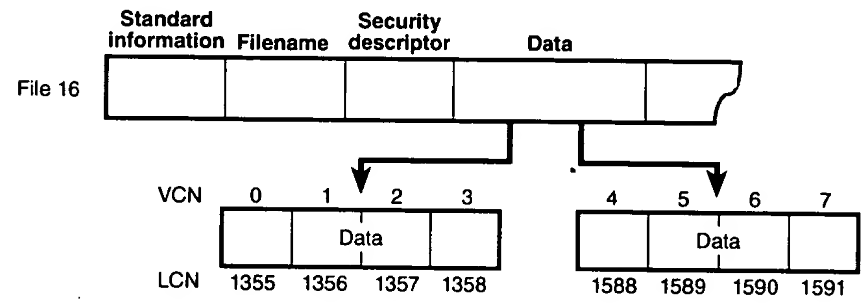


**Figure 9-15**
*VCNs for a nonresident data attribute*

If this file had more than two runs, the numbering of the third run would start with VCN 8. As Figure 9-16 shows, the data attribute header contains VCN-to-LCN mappings for the two runs here, which allows NTFS to easily find the allocations on the disk.

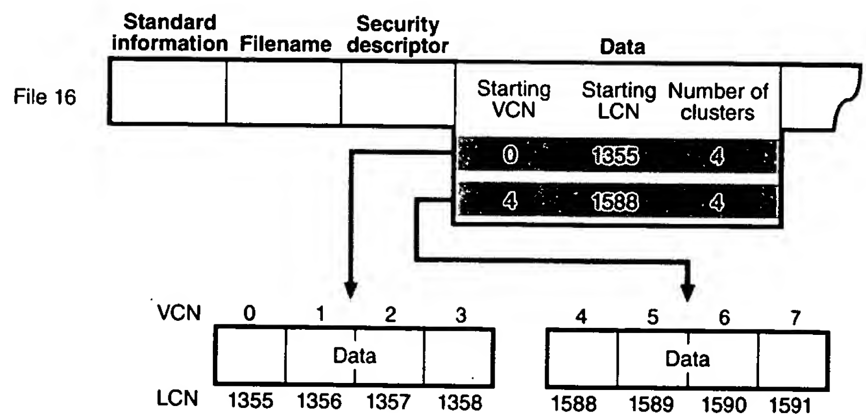


**Figure 9-16**
*VCN-to-LCN mappings for a nonresident data attribute*

Although Figure 9-16 shows just data runs, other attributes can be stored in runs if there isn't enough room in the MFT file record to contain them. And if a particular file has too many attributes to fit in the MFT record, a second MFT record is used to contain the additional attributes (or attribute headers for nonresident attributes). In this case, an attribute called the *attribute list* is added. The attribute list attribute contains the name and type code of each of the file's attributes and the file reference of the MFT record where the attribute is located. The attribute list attribute is provided for those cases in which a file grows so large or so fragmented that a single MFT record can't contain the multitude of VCN-to-LCN mappings needed to find all of its runs. NTFS needs this attribute so rarely that special dysfunctional programs had to be written to test the NTFS code that implements attribute lists.

Windows NT 5.0 **NOTE** With Windows NT 5.0, NTFS will support efficient allocation of sparse files, that is, files that might contain large amounts of unused (either zero or undefined) space.

## Filename Indexing

In NTFS, a file directory is simply an index of filenames—that is, a collection of filenames (along with their file references) organized in a particular way for quick access. To create a directory, NTFS indexes the filename attributes of the files in the directory. The MFT record for the root directory of a volume is shown in Figure 9-17.

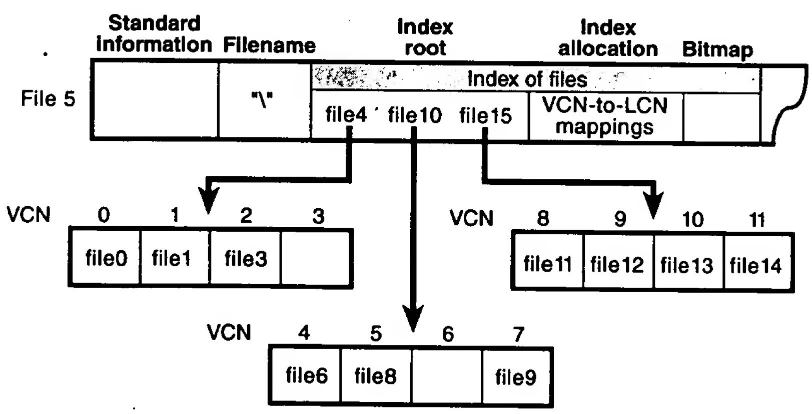


**Figure 9-17**
*Filename index for a volume's root directory*

Conceptually, an MFT entry for a directory contains in its index root attribute a sorted list of the files in the directory. For large directories, however, the filenames are actually stored in 4-KB fixed-size index buffers that contain and organize the filenames. Index buffers implement a *b+ tree* data structure, which minimizes the number of disk accesses needed to find a particular file, especially for large directories. The index root attribute contains the first level of the b+ tree (root subdirectories) and points to index buffers containing the next level (more subdirectories, perhaps, or files). The index allocation attribute maps the VCNs of the index buffer runs to the LCNs that indicate where the index buffers reside on the disk.

Figure 9-17 on the preceding page shows only filenames in the index root attribute and the index buffers (*file6*, for example), but each entry in an index also contains the file reference in the MFT where the file is described and time stamp and file size information for the file. NTFS duplicates the time stamp and file size information from the file's MFT record. This technique, which is used by FAT and NTFS, requires updated information to be written in two places. Even so, it's a significant speed optimization for directory browsing because it enables the file system to display each file's time stamps and size without opening every file in the directory.

The index allocation attribute contains the VCN-to-LCN mappings for the index buffers, and the bitmap attribute keeps track of which VCNs in the index buffers are in use and which are free. Figure 9-17 shows one file entry per VCN (that is, per cluster), but filename entries are actually packed into each cluster. Each 4-KB index buffer can contain about 20 to 30 filename entries.

The b+ tree data structure is a type of balanced tree that is ideal for organizing sorted data stored on a disk because it minimizes the number of disk accesses needed to find an entry. In the MFT, a directory's index root attribute contains several filenames that act as indexes into the second level of the b+ tree. Each filename in the index root attribute has an optional pointer associated with it that points to an index buffer. The index buffer it points to contains filenames with lexicographic values less than its own. In Figure 9-17, for example, *file4* is a first-level entry in the b+ tree. It points to an index buffer containing filenames that are (lexicographically) less than itself—the filenames *file0*, *file1*, and *file3*.

Storing the filenames in b+ trees provides several benefits. Directory lookups are fast because the filenames are stored in a sorted order. And when higher-level software enumerates the files in a directory, NTFS returns already-sorted names. Finally, because b+ trees tend to grow wide rather than deep, NTFS's fast lookup times don't degrade as directories get large.

NTFS currently indexes only the filename attribute, but as noted earlier, NTFS in Windows NT 5.0 will index other file attributes.

## Data Compression

NTFS supports compression on a per-file, per-directory, or per-volume basis. (Currently, NTFS compression is performed only on user data, not file system metadata.) You can tell if a volume is compressed by using the Win32 *GetVolumeInformation* function. To retrieve the actual compressed size of a file, use the Win32 *GetCompressedFileSize* function. Finally, to examine or change the compression setting for a file or directory, use the Win32 *DeviceIoControl* function. (See the FSCTL_GET_COMPRESSION and FSCTL_SET_COMPRESSION I/O function codes.) Keep in mind that although setting a file's compression state compresses (or decompresses) the file right away, setting a directory's compression state doesn't cause any immediate compression or decompression. Instead, setting a directory's compression state sets a default compression state that will be given to all newly created files and subdirectories.

The following section introduces NTFS compression by examining the simple case of compressing sparse files. The subsequent section extends the discussion to the compression of ordinary files.

### Compressing a Sparse File

*Sparse files* are files, often large, that contain only a small amount of nonzero data relative to their size. A sparse matrix stored on disk is one example of a sparse file.

> **NOTE** In this section, sparse files do not refer to the upcoming Windows NT 5.0 enhancement to eliminate allocation of unused or empty space in sparse files.

NTFS uses VCNs, from 0 through *m*, to enumerate the clusters of a file. Each VCN maps to a corresponding LCN, which identifies the disk location of the cluster. Figure 9-18 illustrates the runs (disk allocations) of a normal, noncompressed file, including its VCNs and the LCNs they map to.

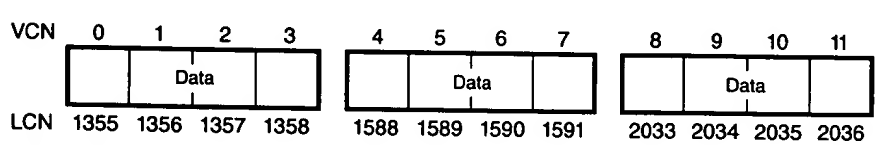


**Figure 9-18**
*Runs of a noncompressed file*

This file is stored in 3 runs, each of which is 4 clusters long, for a total of 12 clusters. Figure 9-19 shows the MFT record for this file. To save space, the MFT record's data attribute, which contains VCN-to-LCN mappings, records only one mapping for each run, rather than one for each cluster. Notice, however, that each VCN from 0 through 11 has a corresponding LCN associated with it. The first entry starts at VCN 0 and covers 4 clusters, the second entry starts at VCN 4 and covers 4 clusters, and so on. This entry format is typical for a noncompressed file.

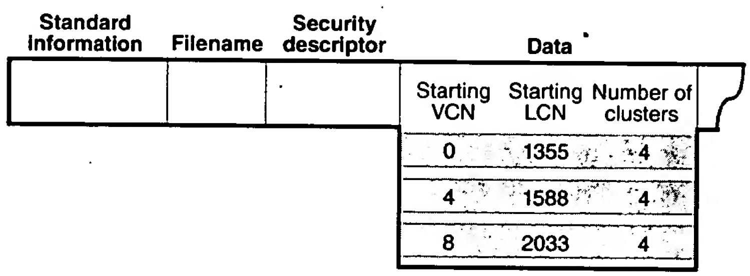


**Figure 9-19**
*MFT record for a noncompressed file*

When a user selects a file on an NTFS volume for compression, one NTFS compression technique is to remove long strings of zeros from the file. If the file is sparse, it typically shrinks to occupy a fraction of the disk space it would otherwise require. On subsequent writes to the file, NTFS allocates space only for runs that contain nonzero data.

Figure 9-20 depicts the runs of a compressed sparse file. Notice that certain ranges of the file's VCNs (16–31 and 64–127) have no disk allocations.

The MFT record for this sparse file omits blocks of VCNs that contain zeros and therefore have no physical storage allocated to them. The first data entry in Figure 9-21, for example, starts at VCN 0 and covers 16 clusters. The second entry jumps to VCN 32 and covers 16 clusters.

When a program reads data from a compressed file, NTFS checks the MFT record to determine whether a VCN-to-LCN mapping covers the location being read. If the program is reading from an unallocated "hole" in the file, it means that the data in that part of the file consists of zeros, so NTFS returns zeros without accessing the disk. If a program writes nonzero data to a "hole," NTFS quietly allocates disk space and then writes the data. This technique is very efficient for sparse files that contain a lot of zero data.

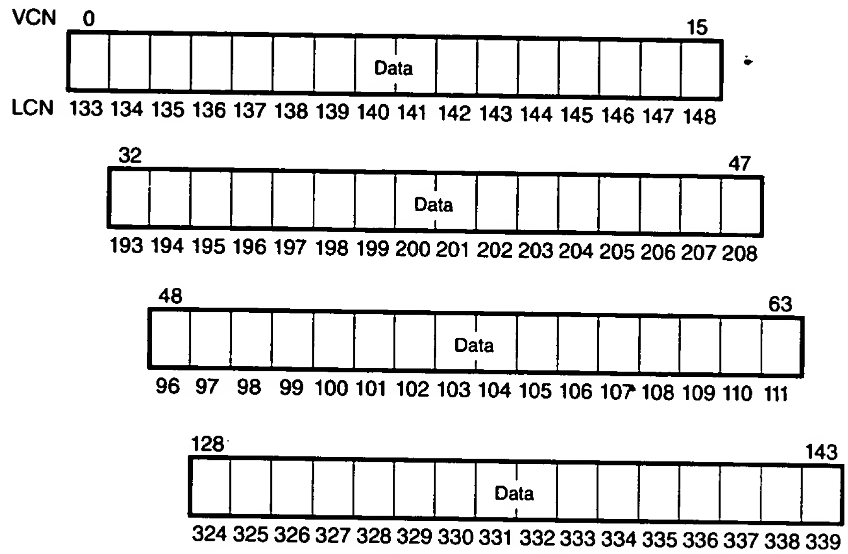


**Figure 9-20**
*Runs of a compressed sparse file*

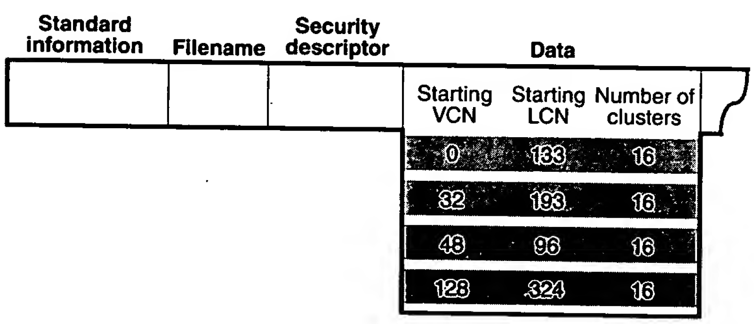


**Figure 9-21**
*MFT record for a compressed sparse file*

## Compressing Nonsparse Data

The preceding example of compressing a sparse file is somewhat contrived. It describes "compression" for a case in which whole sections of a file were filled with zeros but the remaining data in the file wasn't affected by the compression. The data in most files is not sparse, but it can still be compressed by the application of a compression algorithm.

In NTFS, users can specify compression for individual files or for all the files in a directory. When it compresses a file, NTFS divides the file's unprocessed data into *compression units* 16 clusters long (equal to 8 KB for a 512-byte cluster). Certain sequences of data in a file might not compress much, if at all; so for each compression unit in the file, NTFS determines whether compressing the unit will save at least 1 cluster of storage. If compressing the unit won't free up at least 1 cluster, NTFS allocates a 16-cluster run and writes the data in that unit to disk without compressing it. If the data in a 16-cluster unit will compress to 15 or fewer clusters, NTFS allocates only the number of clusters needed to contain the compressed data and then writes it to disk. Figure 9-22 illustrates the compression of a file with four runs. The unshaded areas in this figure represent the actual storage locations that the file occupies after compression. The first, second, and fourth runs were compressed; the third run was not. Even with one noncompressed run, compressing this file saved 26 clusters of disk space, or 41 percent.

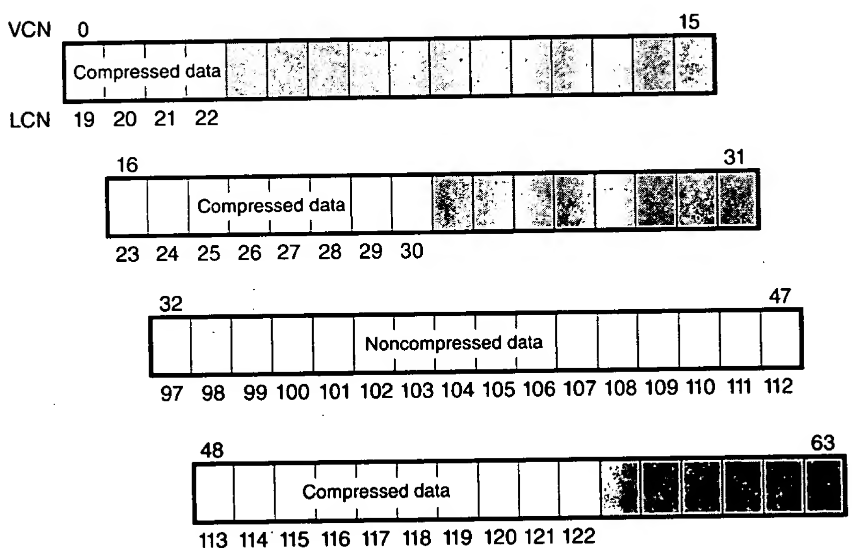


**Figure 9-22**
*Data runs of a compressed file*

**NOTE** Although the diagrams in this chapter show contiguous LCNs, a compression unit need not be stored in physically contiguous clusters. Runs that occupy noncontiguous clusters produce slightly more complicated MFT records than the one shown in Figure 9-23.

When it writes data to a compressed file, NTFS ensures that each run begins on a virtual 16-cluster boundary. Thus the starting VCN of each run is a multiple of 16, and the runs are no longer than 16 clusters. NTFS reads and writes at least one compression unit at a time when it accesses compressed files. When it writes compressed data, however, NTFS tries to store compression units in physically contiguous locations so that it can read them all in a single I/O operation. The 16-cluster size of the NTFS compression unit was chosen to reduce internal fragmentation: the larger the compression unit, the less the overall disk space needed to store the data. This 16-cluster compression unit size represents a trade-off between producing smaller compressed files and slowing read operations for programs that randomly access files. The equivalent of 16 clusters must be decompressed for each cache miss. (A cache miss is more likely to occur during random file access.) Figure 9-23 shows the MFT record for the compressed file shown in Figure 9-22.

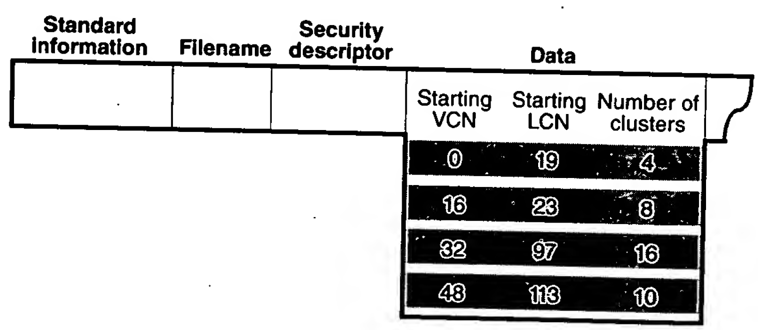


**Figure 9-23**
*MFT record for a compressed file*

One difference between this compressed file and the earlier example of a compressed sparse file is that three of the compressed runs in this file are less than 16 clusters long. Reading this information from a file's MFT file record enables NTFS to know whether data in the file is compressed. Any run shorter than 16 clusters contains compressed data that NTFS must decompress when it first reads the data into the cache. A run that is exactly 16 clusters long doesn't contain compressed data and therefore requires no decompression.

If the data in a run has been compressed, NTFS decompresses the data into a scratch buffer and then copies it to the caller's buffer. NTFS also loads the decompressed data into the cache, which makes subsequent reads from the same run as fast as any other cached read. NTFS writes any updates to the file

in the cache, leaving the lazy writer to compress and write the modified data to disk asynchronously. This strategy ensures that writing to a compressed file produces no more significant delay than writing to a noncompressed file would.

NTFS keeps disk allocations for a compressed file contiguous whenever possible. As the LCNs indicate, the first two runs of the compressed file shown in Figure 9-22 on page 424 are physically contiguous, as are the last two. When two or more runs are contiguous, NTFS performs disk read-ahead, as it does with the data in other files. Because the reading and decompression of contiguous file data take place asynchronously before the program requests the data, subsequent read operations obtain the data directly from the cache, which greatly enhances read performance.

## Recoverability Support

NTFS recovery support ensures that if a power failure or a catastrophic system failure occurs, no file system operations (transactions) will be left incomplete and the structure of the disk volume will remain intact without the need to run a disk repair utility. The NTFS Chkdsk utility is used to repair catastrophic disk corruption caused by I/O errors (bad disk sectors, electrical anomalies, or disk failures, for example) or software bugs. But with the NTFS recovery capabilities in place, Chkdsk is rarely needed.

NTFS uses a transaction-based logging scheme to implement recoverability. This strategy ensures a full disk recovery that is also extremely fast (on the order of seconds) for even the largest disks. NTFS limits its recovery procedures to file system data to ensure that at the very least the user will never lose a volume because of a corrupted file system; however, user data is not guaranteed to be fully updated if a crash occurs. Transaction-based protection of user data is available in most of the database products available for Windows NT, such as Microsoft SQL Server. The decision not to implement user data recovery in the file system represents a trade-off between a fully fault tolerant file system and one that provides optimum performance for all file operations.

The following sections describe the evolution of file system reliability as a context for an introduction to recoverable file systems, detail the transaction-logging scheme NTFS uses to record modifications to file system data structures, and explain how NTFS recovers a volume if the system fails.

### Evolution of File System Design

The development of a recoverable file system is a step forward in the evolution of file system design. In the past, two techniques were common for constructing a file system's I/O and caching support: *careful write* and *lazy write*. The file

systems developed for Digital Equipment Corporation's VAX/VMS and for some other proprietary operating systems employed a careful write algorithm, while OS/2 HPFS and most older UNIX file systems used a lazy write file system scheme.

The next two subsections briefly review these two types of file systems and their intrinsic trade-offs between safety and performance. The third subsection describes NTFS's recoverable approach and explains how it differs from the two other strategies.

### Careful Write File Systems

When an operating system crashes or loses power, I/O operations in progress are immediately, and often prematurely, interrupted. Depending on what I/O operation or operations were in progress and how far along they were, such an abrupt halt can produce inconsistencies in a file system. An inconsistency in this context is a file system corruption—a filename appears in a directory listing, for example, but the file system doesn't know the file is there or can't access the file. The worst file system corruptions can leave an entire volume inaccessible.

A careful write file system doesn't try to prevent file system inconsistencies. Rather, it orders its write operations so that, at worst, a system crash will produce predictable, noncritical inconsistencies, which the file system can fix at its leisure.

When any kind of file system receives a request to update the disk, it must perform several suboperations before the update will be complete. In a file system that uses the careful write strategy, the suboperations are always written to disk serially. When allocating disk space for a file, for example, the file system first sets some bits in its bitmap and then allocates the space to the file. If the power fails immediately after the bits are set, the careful write file system loses access to some disk space—to the space represented by the set bits—but existing data is not corrupted.

Serializing write operations also means that I/O requests are filled in the order in which they are received. If one process allocates disk space and shortly thereafter another process creates a file, a careful write file system completes the disk allocation before it starts to create the file because interleaving the suboperations of the two I/O requests could result in an inconsistent state.

> **NOTE** The FAT file system uses a *write-through* algorithm that causes disk modifications to be immediately written to the disk. Unlike the careful write approach, the write-through technique doesn't require the file system to order its writes to prevent inconsistencies.

The main advantage of a careful write file system is that in the event of a failure the volume stays consistent and usable without the need to immediately run a slow volume repair utility. Such a utility is needed to correct the predictable, nondestructive disk inconsistencies that occur as the result of a system failure, but the utility can be run at a convenient time, typically when the system is rebooted.

### Lazy Write File Systems

A careful write file system sacrifices speed for the safety it provides. A lazy write file system improves performance by using a *write-back* caching strategy; that is, it writes file modifications to the cache and flushes the contents of the cache to disk in an optimized way, usually as a background activity.

The performance improvements associated with the lazy write caching technique take several forms. First, the overall number of disk writes is reduced. Because serialized, immediate disk writes aren't required, the contents of a buffer can be modified several times before they are written to disk. Second, the speed of servicing application requests is greatly increased because the file system can return control to the caller without waiting for disk writes to be completed. Finally, the lazy write strategy ignores the inconsistent intermediate states on a file volume that can result when the suboperations of two or more I/O requests are interleaved. It is thus easier to make the file system multithreaded, allowing more than one I/O operation to be in progress at a time.

The disadvantage of the lazy write technique is that it creates intervals during which a volume is in too inconsistent a state to be corrected by the file system. Consequently, lazy write file systems must keep track of the volume's state at all times. In general, lazy write file systems gain a performance advantage over careful write systems—at the expense of greater risk and user inconvenience if the system fails.

### Recoverable File Systems

A recoverable file system tries to exceed the safety of a careful write file system while achieving the performance of a lazy write file system. A recoverable file system ensures volume consistency by using logging techniques originally developed for transaction processing. If the operating system crashes, the recoverable file system restores consistency by executing a recovery procedure that accesses information that has been stored in a log file. Because the file system has logged its disk writes, the recovery procedure takes only seconds, regardless of the size of the volume.

The NTFS recovery procedure is exact, guaranteeing that the volume will be restored to a consistent state. None of the inadequate restorations associated with lazy write file systems can happen with NTFS.

A recoverable file system incurs some costs for the safety it provides. Every transaction that alters the volume structure requires that one record be written to the log file for each of the transaction's suboperations. This logging overhead is ameliorated by the file system's "batching" of log records—writing many records to the log file in a single I/O operation. In addition, the recoverable file system can employ the optimization techniques of a lazy write file system. It can even increase the length of the intervals between cache flushes because the file system can be recovered if the system crashes before the cache changes have been flushed to disk. This gain over the caching performance of lazy write file systems makes up for, and often exceeds, the overhead of the recoverable file system's logging activity.

Neither careful write nor lazy write file systems guarantee protection of user file data. If the system crashes while an application is writing a file, the file can be lost or corrupted. Worse, the crash can corrupt a lazy write file system, destroying existing files or even rendering an entire volume inaccessible.

NTFS implements several strategies that improve its reliability over that of the traditional file systems. First, NTFS recoverability guarantees that the volume structure won't be corrupted, so all files will remain accessible after a system failure.

Second, although NTFS doesn't currently guarantee protection of user data in the event of a system crash—some changes can be lost from the cache—applications can take advantage of the NTFS write-through and cache-flushing capabilities to ensure that file modifications are recorded on disk at appropriate intervals. Both *cache write-through*—forcing write operations to be immediately recorded on disk—and *cache flushing*—forcing cache contents to be written to disk—are efficient operations. NTFS doesn't have to do extra disk I/O to flush modifications to several different file system data structures because changes to the data structures are recorded—in a single write operation—in the log file; if a failure occurs and cache contents are lost, the file system modifications can be recovered from the log. Furthermore, unlike the FAT file system, NTFS guarantees that user data will be consistent and available immediately after a write-through operation or a cache flush, even if the system subsequently fails.

Finally, NTFS has all the underpinnings to support logging for user files in the future. In lieu of user data logging, users who require an added measure of data reliability can use FtDisk, the Windows NT fault tolerant disk driver, to set up and maintain redundant data storage. (See the section "Fault Tolerance Support" later in this chapter for more information about data redundancy.)

## Logging

NTFS provides file system recoverability by means of a transaction-processing technique called *logging*. In NTFS logging, the suboperations of any transaction that alters important file system data structures are recorded in a log file before they are carried through on the disk. That way, if the system crashes, partially completed transactions can be redone or undone when the system comes back online. In transaction processing, this technique is known as *write-ahead logging*. In NTFS, transactions include writing to the disk or deleting a file and can be made up of several suboperations.

### Log File Service (LFS)

LFS is a series of kernel-mode routines inside the NTFS driver that NTFS uses to access the log file. Although originally designed to provide logging and recovery services for more than one client, LFS is currently used only by NTFS. The caller—NTFS in this case—passes the LFS a pointer to an open file object, which specifies a log file to be accessed. The LFS either initializes a new log file or calls the Windows NT cache manager to access the existing log file through the cache, as shown in Figure 9-24.

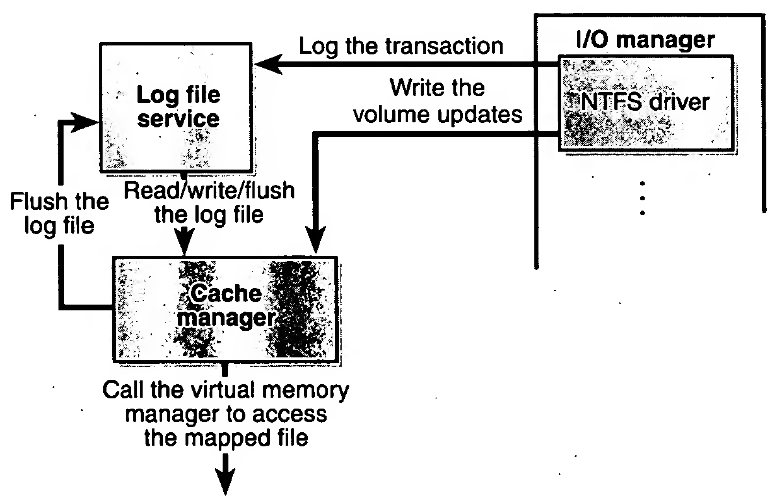


**Figure 9-24.**
*Log file service (LFS)*

The LFS divides the log file into two regions: a *restart area* and an "infinite" *logging area,* as shown in Figure 9-25.

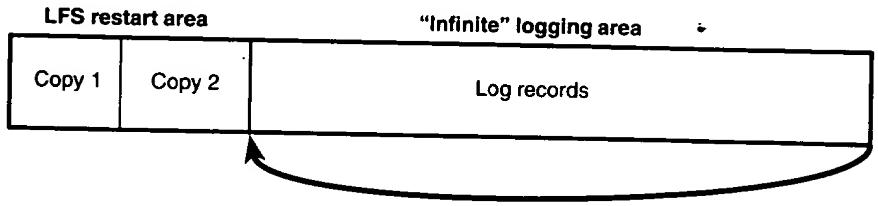


**Figure 9-25**
*Log file regions*

NTFS calls the LFS to read and write the restart area. NTFS uses the restart area to store context information such as the location in the logging area at which NTFS will begin to read during recovery after a system failure. The LFS maintains a second copy of the restart data in case the first becomes corrupted or otherwise inaccessible. The remainder of the log file is the logging area, which contains transaction records NTFS writes in order to recover a volume in the event of a system failure. The LFS makes the log file appear infinite by reusing it circularly (while guaranteeing that it doesn't overwrite information it needs). The LFS uses *logical sequence numbers* (LSNs) to identify records written to the log file. As the LFS cycles through the file, it increases the values of the LSNs. The number of possible LSNs is so large as to be virtually infinite.

NTFS never reads transactions from or writes transactions to the log file directly. The LFS provides services NTFS calls to open the log file, write log records, read log records in forward or backward order, flush log records up to a particular LSN, or set the beginning of the log file to a higher LSN. During recovery, NTFS calls the LFS to read forward through the log records in order to redo any transactions that were recorded in the log file but were not flushed to disk at the time of the system failure. NTFS calls the LFS to read backward through the log records in order to undo, or roll back, any transactions that weren't completely logged before the crash. NTFS calls the LFS to set the beginning of the log file to a record with a higher LSN when NTFS no longer needs the older transaction records in the log file.

Here's how the system guarantees that the volume can be recovered:

1. NTFS first calls the LFS to record in the (cached) log file any transactions that will modify the volume structure.
2. NTFS modifies the volume (also in the cache).

3. The cache manager calls the LFS to prompt the LFS to flush the log file to disk. (The LFS implements the flush by calling the cache manager back, telling it which pages of memory to flush. Refer back to the calling sequence shown in Figure 9-24 on page 430.)
4. After the cache manager flushes the log file to disk, it flushes the volume changes (the transactions themselves) to disk.

These steps ensure that if the file system modifications are ultimately unsuccessful, the corresponding transactions can be retrieved from the log file and can be either redone or undone as part of the file system recovery procedure.

File system recovery begins automatically the first time the volume is used after the system is rebooted. NTFS checks whether the transactions that were recorded in the log file before the crash were applied to the volume, and if they weren't, it redoes them. NTFS also guarantees that transactions not completely logged before the crash are undone so that they don't appear on the volume.

## Log Record Types

The LFS allows its clients to write any kind of record to their log files. NTFS writes several types of records. Two types, *update records* and *checkpoint records,* are described here.

**Update records** Update records are the most common type of record NTFS writes to the log file. Each update record contains two kinds of information:

- **Redo information** How to reapply one suboperation of a fully logged ("committed") transaction to the volume if a system failure occurs before the transaction is flushed from the cache
- **Undo information** How to reverse one suboperation of a transaction that was only partially logged ("not committed") at the time of a system failure

Figure 9-26 shows three update records in the log file. Each record represents one suboperation of a transaction, creating a new file. The redo entry in each update record tells NTFS how to reapply the suboperation to the volume, and the undo entry tells NTFS how to roll back (undo) the suboperation.

After logging a transaction (in this example, by calling the LFS to write the three update records to the log file), NTFS performs the suboperations on the volume itself, in the cache. When it has finished updating the cache, NTFS

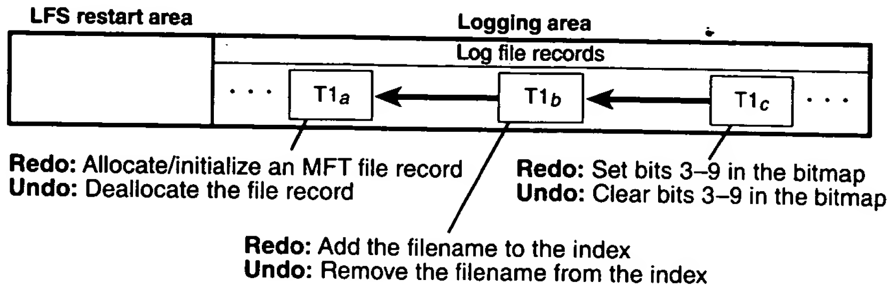


**Figure 9-26**
*Update records in the log file*

writes another record to the log file, recording the entire transaction as complete—a suboperation known as *committing a transaction.* Once a transaction is committed, NTFS guarantees that the entire transaction will appear on the volume, even if the operating system subsequently fails.

When recovering after a system failure, NTFS reads through the log file and redoes each committed transaction. Although NTFS completed the committed transactions before the system failure, it doesn't know whether the cache manager flushed the volume modifications to disk in time. The updates might have been lost from the cache when the system failed. Therefore, NTFS executes the committed transactions again just to be sure that the disk is up to date.

After redoing the committed transactions during a file system recovery, NTFS locates all the transactions in the log file that were not committed at failure and rolls back (undoes) each suboperation that had been logged. In Figure 9-26, NTFS would first undo the $T1_c$ suboperation and then follow the backward pointer to $T1_b$ and undo that suboperation. It would continue to follow the backward pointers, undoing suboperations, until it reached the first suboperation in the transaction. By following the pointers, NTFS knows how many and which update records it must undo to roll back a transaction.

Redo and undo information can be expressed either physically or logically. Physical descriptions specify volume updates in terms of particular byte ranges on the disk that are to be changed, moved, and so on. Logical descriptions express updates in terms of operations such as "delete file A.DAT." As the lowest layer of software maintaining the file system structure, NTFS writes update records with physical descriptions. Transaction-processing or other application-level software might benefit from writing update records in logical terms, however, because logically expressed updates are more compact than

physically expressed ones. Logical descriptions necessarily depend on NTFS to understand what operations, such as deleting a file, involve.

NTFS writes update records (usually several) for each of the following transactions:

- Creating a file
- Deleting a file
- Extending a file
- Truncating a file
- Setting file information
- Renaming a file
- Changing the security applied to a file

The redo and undo information in an update record must be carefully designed because although NTFS undoes a transaction, recovers from a system failure, or even operates normally, it might try to redo a transaction that has already been done or, conversely, to undo a transaction that never occurred or that has already been undone. Similarly, NTFS might try to redo or undo a transaction consisting of several update records, only some of which are complete on disk. The format of the update records must ensure that executing redundant redo or undo operations is *idempotent,* that is, has a neutral effect. For example, setting a bit that is already set has no effect, but toggling a bit that has already been toggled does. The file system must also handle intermediate volume states correctly.

**Checkpoint records** In addition to update records, NTFS periodically writes a checkpoint record to the log file, as illustrated in Figure 9-27.

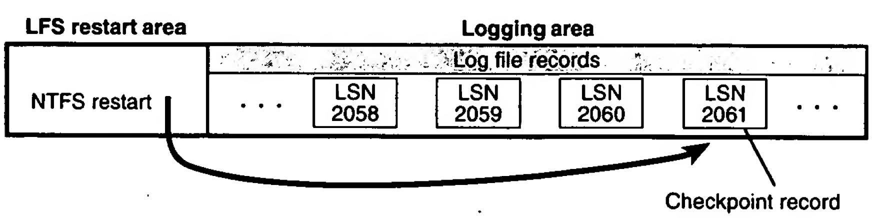


**Figure 9-27**
*Checkpoint record in the log file*

A checkpoint record helps NTFS determine what processing would be needed to recover a volume if a crash were to occur immediately. Using information stored in the checkpoint record, NTFS knows, for instance, how far back in the log file it must go to begin its recovery. After writing a checkpoint record, NTFS stores the LSN of the record in the restart area so that it can quickly find its most recently written checkpoint record when it begins file system recovery after a crash occurs.

Although the LFS presents the log file to NTFS as if it were infinitely large, it isn't. The generous size of the log file and the frequent writing of checkpoint records (an operation that usually frees up space in the log file) make the possibility of the log file's filling up a remote one. Nevertheless, the LFS accounts for this possibility by tracking several numbers:

- The available log space
- The amount of space needed to write an incoming log record and to undo the write, should that be necessary
- The amount of space needed to roll back all active (noncommitted) transactions, should that be necessary

If the log file doesn't contain enough available space to accommodate the total of the last two items, the LFS returns a "log file full" error and NTFS raises an exception. The NTFS exception handler rolls back the current transaction and places it in a queue to be restarted later.

To free up space in the log file, NTFS must momentarily prevent further transactions on files. To do so, NTFS blocks file creation and deletion and then requests exclusive access to all system files and shared access to all user files. Gradually, active transactions either are completed successfully or receive the "log file full" exception. NTFS rolls back and queues the transactions that receive the exception.

Once it has blocked transaction activity on files as described above, NTFS calls the cache manager to flush unwritten data to disk, including unwritten log file data. After everything is safely flushed to disk, NTFS no longer needs the data in the log file. It resets the beginning of the log file to the current position, making the log file "empty." Then it restarts the queued transactions. Beyond the short pause in I/O processing, the "log file full" error has no effect on executing programs.

This scenario is one example of how NTFS uses the log file not only for file system recovery but also for error recovery during normal operation. You'll find out more about error recovery in the following section.

## Recovery

NTFS automatically performs a disk recovery the first time a program accesses an NTFS volume after the system has been booted. (If no recovery is needed, the process is trivial.) Recovery depends on two tables NTFS maintains in memory:

- The *transaction table* keeps track of transactions that have been started but are not yet committed. The suboperations of these active transactions must be removed from the disk during recovery.
- The *dirty page table* records which pages in the cache contain modifications to the file system structure that have not yet been written to disk. This data must be flushed to disk during recovery.

NTFS writes a checkpoint record to the log file once every 5 seconds. Just before it does, it calls the LFS to store a current copy of the transaction table and of the dirty page table in the log file. NTFS then records in the checkpoint record the LSNs of the log records containing the copied tables. When recovery begins after a system failure, NTFS calls the LFS to locate the log records containing the most recent checkpoint record and the most recent copies of the transaction and dirty page tables. It then copies the tables to memory.

The log file usually contains more update records following the last checkpoint record. These update records represent volume modifications that occurred after the last checkpoint record was written. NTFS must update the transaction and dirty page tables to include these operations. After updating the tables, NTFS uses the tables and the contents of the log file to update the volume itself.

To effect its volume recovery, NTFS scans the log file three times, loading the file into memory during the first pass to minimize disk I/O. Each pass has a particular purpose:

1. Analysis
2. Redoing transactions
3. Undoing transactions

### Analysis Pass

During the *analysis pass,* as shown in Figure 9-28, NTFS scans forward in the log file from the beginning of the last checkpoint operation in order to find update records and use them to update the transaction and dirty page tables it copied to memory. Notice in the figure that the checkpoint operation stores

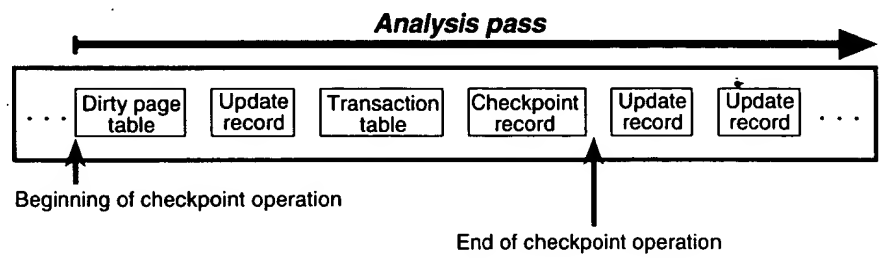


**Figure 9-28**
*Analysis pass*

three records in the log file and that update records might be interspersed among these records. NTFS therefore must start its scan at the beginning of the checkpoint operation.

Each update record that appears in the log file after the checkpoint operation begins represents a modification to either the transaction table or the dirty page table. If an update record is a "transaction committed" record, for example, the transaction the record represents must be removed from the transaction table. Similarly, if the update record is a "page update" record that modifies a file system data structure, the dirty page table must be updated to reflect that change.

Once the tables are up to date in memory, NTFS scans the tables to determine the LSN of the oldest update record that logs an operation that has not been carried out on disk. The transaction table contains the LSNs of the noncommitted (incomplete) transactions, and the dirty page table contains the LSNs of records in the cache that have not been flushed to disk. The LSN of the oldest record that NTFS finds in these two tables determines where the redo pass will begin. If the last checkpoint record is older, however, NTFS will start the redo pass there instead.

### Redo Pass

During the *redo pass,* as shown in Figure 9-29 on the following page, NTFS scans forward in the log file from the LSN of the oldest record it has found in the analysis pass. It looks for "page update" records, which contain volume modifications that were written before the system failure but that might not have been flushed to disk. NTFS redoes these updates in the cache.

When NTFS reaches the end of the log file, it has updated the cache with the necessary volume modifications and the cache manager's lazy writer can begin writing cache contents to disk in the background.

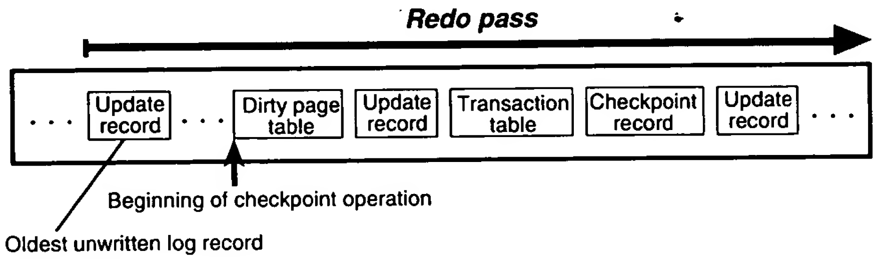


**Figure 9-29**
*Redo pass*

## Undo Pass

After it completes the redo pass, NTFS begins its *undo pass,* in which it rolls back any transactions that weren't committed when the system failed. Figure 9-30 shows two transactions in the log file; transaction 1 was committed before the power failure, but transaction 2 was not. NTFS must undo transaction 2.

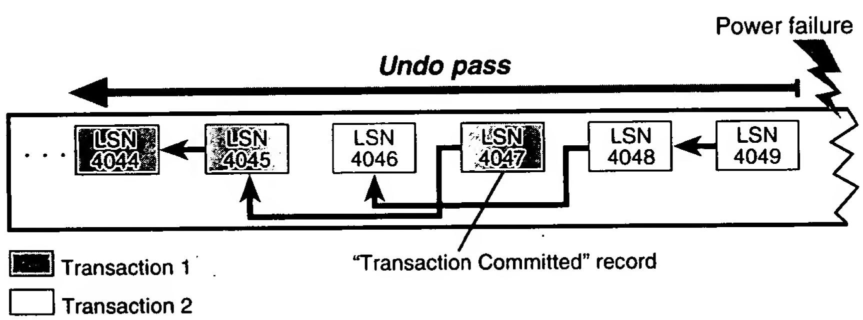


**Figure 9-30**
*Undo pass*

Suppose that transaction 2 created a file, an operation that comprises three suboperations, each with its own update record. The update records of a transaction are linked by backward pointers in the log file because they are usually not contiguous.

The NTFS transaction table lists the LSN of the last-logged update record for each noncommitted transaction. In this example, the transaction table identifies LSN 4049 as the last update record logged for transaction 2. As shown from right to left in Figure 9-31, NTFS rolls back transaction 2.

Each update record contains two kinds of information: how to redo a suboperation and how to undo it. After locating LSN 4049, NTFS finds the undo information and executes it, clearing bits 3 through 9 in its allocation bitmap. NTFS then follows the backward pointer to LSN 4048, which directs it

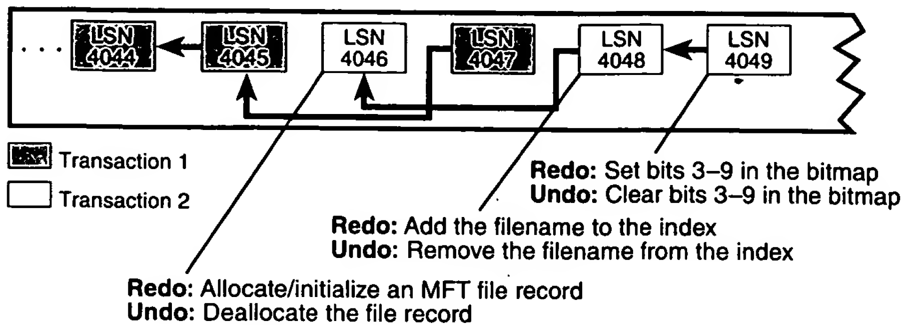


**Figure 9-31**
*Undoing a transaction*

to remove the new filename from the appropriate filename index. Finally, it follows the last backward pointer and deallocates the MFT file record reserved for the file, as the update record with the LSN 4046 specifies. Transaction 2 is now rolled back. If there are other noncommitted transactions to undo, NTFS follows the same procedure to roll them back. Because undoing transactions affects the volume's file system structure, NTFS must log the undo operations in the log file. After all, the power might fail again during the recovery, and NTFS would have to redo its undo operations!

When the undo pass of the recovery is complete, the volume has been restored to a consistent state. At this point, NTFS flushes the cache changes to disk to ensure that the volume is up to date. NTFS then writes an "empty" LFS restart area to indicate that the volume is consistent and that no recovery need be done if the system should fail again immediately. Recovery is complete.

NTFS guarantees that recovery will return the volume to some preexisting consistent state, but not necessarily to the state that existed just before the system crash. NTFS can't make that guarantee because, for performance, it uses a "lazy commit" algorithm, which means that the log file is not immediately flushed to disk each time a "transaction committed" record is written. Instead, numerous transaction committed records are batched and written together, either when the cache manager calls the LFS to flush the log file to disk or when the LFS writes a checkpoint record (once every 5 seconds) to the log file. Another reason the recovered volume might not be completely up to date is that several parallel transactions might be active when the system crashes and some of their transaction committed records might make it to disk whereas others might not. The consistent volume that recovery produces includes all the volume updates whose transaction committed records made it to disk and none of the updates whose transaction committed records didn't make it to disk.

NTFS uses the log file to recover a volume after the system fails, but it also takes advantage of an important "freebie" it gets from logging transactions. File systems necessarily contain a lot of code devoted to recovering from file system errors that occur during the course of normal file I/O. Because NTFS logs each transaction that modifies the volume structure, it can use the log file to recover when a file system error occurs and thus can greatly simplify its error handling code. The "log file full" error described earlier is one example of using the log file for error recovery.

Most I/O errors a program receives are not file system errors and therefore can't be resolved entirely by NTFS. When called to create a file, for example, NTFS might begin by creating a file record in the MFT and then enter the new file's name in a directory index. When it tries to allocate space for the file in its bitmap, however, it could discover that the disk is full and the create request can't be completed. In such a case, NTFS uses the information in the log file to undo the part of the operation it has already completed and to deallocate the data structures it reserved for the file. Then it returns a "disk full" error to the caller, which in turn must respond appropriately to the error.

# Fault Tolerance Support

The capabilities of NTFS are enhanced by underlying support from a Windows NT driver called FtDisk.sys, the fault tolerant disk driver. FtDisk lies above hard disk drivers in the I/O system's layered driver scheme and provides volume management capabilities, redundant data storage, and dynamic data recovery from bad sectors on SCSI (small computer system interface) disks.

Although FtDisk works with all of the Windows NT–supported file systems, using it with NTFS provides the highest level of data integrity. Even without FtDisk, NTFS removes bad clusters from use and provides the equivalent of FtDisk's bad-sector recovery for non-SCSI hard disks. It also also monitors and detects corruption in file system data structures and uses FtDisk to recover its own data and to ensure its own reliability.

The following two sections describe the volume management and data redundancy capabilities of FtDisk. The third section describes the additional features of NTFS that improve data reliability and recovery.

## Volume Management Features

Although FtDisk is called the fault tolerant driver, it also implements some volume management features unrelated to fault tolerance. Volume sets and stripe sets don't provide data redundancy, but they do aid in organizing volumes and increasing I/O efficiency, respectively.

## Volume Sets

A *volume set* is a single logical volume composed of a maximum of 32 areas of free space on one or more disks. The Windows NT Disk Administrator utility combines the areas into the volume set, which can then be formatted for any of the Windows NT–supported file systems. Figure 9-32 shows a 100-MB volume set identified by drive letter D: that has been created from the last third of the first disk and the first third of the second.

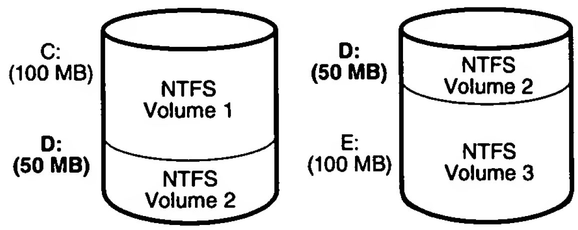


**Figure 9-32**
*Volume set*

A volume set is useful for consolidating small areas of free disk space to create a larger volume or for creating a single, large volume out of two or more small disks. If the volume set has been formatted for NTFS, it can be extended to include additional free areas or additional disks without affecting the data already stored on the volume. This is one of the biggest benefits of describing all data on an NTFS volume as a file. NTFS can dynamically increase the size of a logical volume because the bitmap that records the allocation status of the volume is just another file—the bitmap file. The bitmap file can be extended to include any space added to the volume. Dynamically extending a FAT volume, on the other hand, would require the FAT itself to be extended, which would dislocate everything else on the disk.

FtDisk hides the physical configuration of disks from the file systems installed on Windows NT. NTFS, for example, views D: in Figure 9-32 as an ordinary 100-MB volume. NTFS consults its bitmap to determine what space in the volume is free for allocation. It then calls FtDisk to read or write data beginning at a particular byte offset on the volume. FtDisk views the physical sectors in the volume set as numbered sequentially from the first free area on the first disk to the last free area on the last disk. It determines which physical sector on which disk corresponds to the supplied byte offset.

## Stripe Sets

A *stripe set* is a series of partitions, one partition per disk, that the Disk Administrator utility combines into a single logical volume. Figure 9-33 shows a stripe

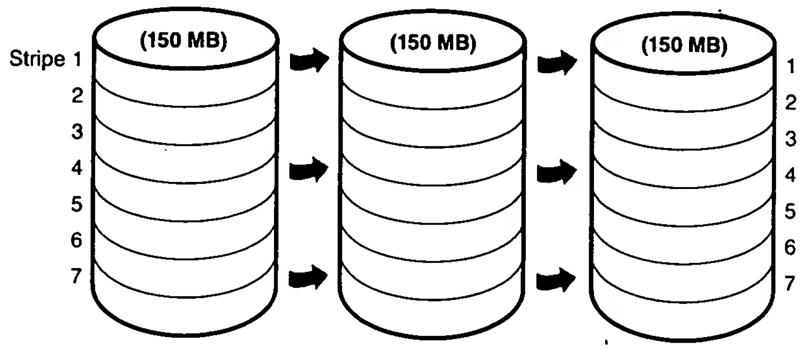


**Figure 9-33**
*Stripe set*

set consisting of three partitions, one on each of three disks. (A partition in a stripe set need not span an entire disk; the only restriction is that the partitions on each disk be the same size.)

To a file system, this stripe set appears to be a single 450-MB volume, but FtDisk optimizes data storage and retrieval times on the stripe set by distributing the volume's data among the physical disks. FtDisk accesses the physical sectors of the disks as if they were numbered sequentially in stripes across the disks, as illustrated in Figure 9-34.

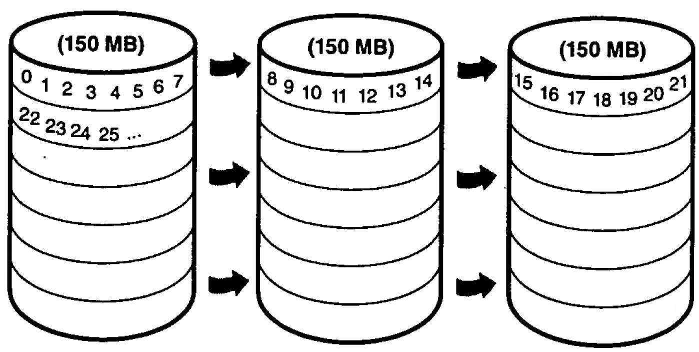


**Figure 9-34**
*Logical numbering of physical sectors on a stripe set*

Because each stripe is a relatively narrow 64 KB (a value chosen to prevent individual reads and writes from accessing two disks), the data tends to be distributed evenly among the disks. Stripes thus increase the probability that

multiple pending read and write operations will be bound for different disks. And because data on all three disks can be accessed simultaneously, latency time for disk I/O is often reduced, particularly on heavily loaded systems.

## Fault Tolerant Volumes

Volume sets make managing disk volumes more convenient, and stripe sets spread the I/O load over multiple disks. These two volume-management features don't provide the ability to recover data if a disk fails, however. For data recovery, FtDisk implements three redundant storage schemes: mirror sets, stripe sets with parity, and sector sparing. These features are created with the Windows NT Disk Administrator utility.

### Mirror Sets

In a *mirror set,* the contents of a partition on one disk are duplicated in an equal-sized partition on another disk. A mirror set is shown in Figure 9-35.

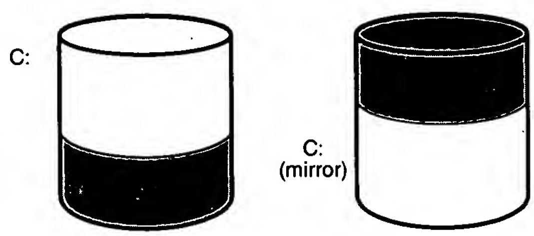


**Figure 9-35**
*Mirror set*

When a program writes to C:, FtDisk writes the same data to the same location on the mirror partition. If the first disk or any of the data on its C: partition becomes unreadable because of a hardware or software failure, FtDisk automatically accesses the data from the mirror partition. A mirror set can be formatted for any of the Windows NT–supported file systems. The file system drivers remain independent and are not affected by FtDisk's mirroring activity.

Mirror sets can aid in I/O throughput on heavily loaded systems. When I/O activity is high, FtDisk balances its read operations between the primary partition and the mirror partition (accounting for the number of unfinished I/O requests pending from each disk). Two read operations can proceed simultaneously and thus theoretically finish in half the time. When a file is modified, both partitions of the mirror set must be written, but disk writes are done asynchronously, so the performance of user-mode programs is generally not affected by the extra disk update.

### Stripe Sets with Parity

A *stripe set with parity* is a fault tolerant variant of a regular stripe set. Fault tolerance is achieved by reserving the equivalent of one disk for storing parity for each stripe. Figure 9-36 is a visual representation of a stripe set with parity.

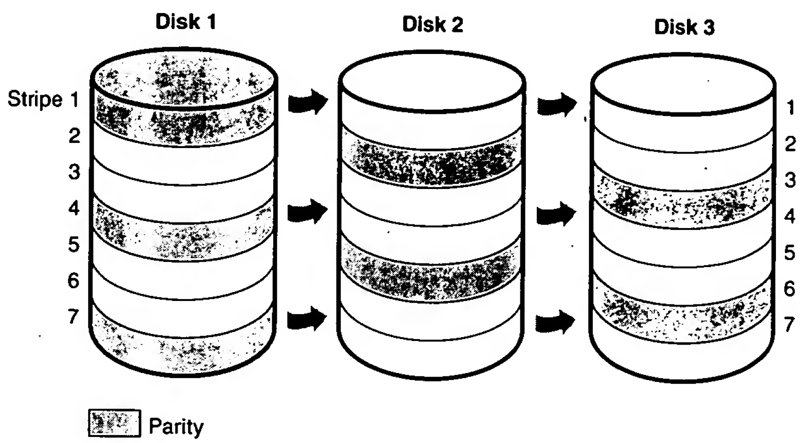


**Figure 9-36**
*Stripe set with parity*

In Figure 9-36, the parity for stripe 1 is stored on disk 1. It contains a byte-for-byte logical sum (XOR) of the first stripe on disks 2 and 3. The parity for stripe 2 is stored on disk 2, and the parity for stripe 3 is stored on disk 3. Rotating the parity across the disks in this way is an I/O optimization technique. Each time data is written to a disk, the parity bytes corresponding to the modified bytes must be recalculated and rewritten. If the parity were always written to the same disk, that disk would be busy continually and could become an I/O bottleneck.

Recovering a failed disk in a stripe set with parity relies on a simple arithmetic principle: in an equation with $n$ variables, if you know the value of $n - 1$ of the variables, you can determine the value of the missing variable by subtraction. For example, in the equation $x + y = z$, where $z$ represents the parity stripe, FtDisk computes $z - y$ to determine the contents of $x$; to find $y$, it computes $z - x$. FtDisk uses similar logic to recover lost data. If a disk in a stripe set with parity fails or if data on one disk becomes unreadable, FtDisk reconstructs the missing data by using the XOR operation (bitwise logical addition).

If disk 1 in Figure 9-36 fails, the contents of its stripes 2 and 5 are calculated by XOR-ing the corresponding stripes of disk 3 with the parity stripes on disk 2. The contents of stripes 3 and 6 are similarly determined by XOR-ing the corresponding stripes of disk 2 with the parity stripes on disk 3. At least three disks (or rather, three same-sized partitions on three disks) are required to create a stripe set with parity.

### Sector Sparing

Redundant data storage is used not only for recovering data after a complete disk failure but also for recovering data from a single physical sector that goes bad. In a technique called *sector sparing,* FtDisk uses its redundant data storage to dynamically replace lost data when a disk sector becomes unreadable. The sector-sparing technique exploits a feature of some hard disks, which provide a set of physical sectors reserved as "spares." If FtDisk receives a data error from the hard disk, it obtains a spare sector from the disk driver to replace the bad sector that caused the data error. FtDisk recovers the data that was on the bad sector (by either reading the data from a disk mirror or recalculating the data from a stripe set with parity) and copies it to the spare sector. FtDisk performs sector sparing dynamically, without intervention from the file system or the user, and sector sparing works with any Windows NT–supported file system on SCSI-based hard disks.

If a bad-sector error occurs and the hard disk doesn't provide spares, runs out of them, or is a non-SCSI-based disk, FtDisk can still recover the data. It recalculates the unreadable data by accessing a stripe set with parity, or it reads a copy of the data from a disk mirror. It then passes the data to the file system along with a warning status that only one copy of the data remains in a disk mirror or that one stripe is inaccessible in a stripe set with parity and that data redundancy is therefore no longer in effect for that sector. It's up to the file system to respond to (or ignore) the warning. FtDisk will re-recover the data each time the file system tries to read from the bad sector.

## NTFS Bad-Cluster Recovery

FtDisk can recover data from a bad sector on a fault tolerant volume, but if the hard disk doesn't use the SCSI protocol or runs out of spare sectors, FtDisk can't perform sector sparing to replace the bad sector. When the file system reads from the sector, FtDisk instead recovers the data and returns the warning to the file system that there is only one copy of the data.

The FAT file system doesn't respond to this FtDisk warning. Moreover, neither these file systems nor FtDisk keeps track of the bad sectors, so a user must run the Chkdsk or Format utility to prevent FtDisk from repeatedly recovering data for the file system. Both Chkdsk and Format are less than ideal for removing bad sectors from use. Chkdsk can take a long time to find and remove bad sectors, and Format wipes all the data off the partition it is formatting.

In the file system equivalent of FtDisk's sector sparing, NTFS dynamically replaces the cluster containing a bad sector and keeps track of the bad cluster so that it won't be reused. (As described earlier, NTFS maintains portability by addressing logical clusters rather than physical sectors.) NTFS performs these functions when FtDisk can't perform sector sparing or when FtDisk is not installed in the system. When FtDisk returns a bad-sector warning or when the hard disk driver returns a bad-sector error, NTFS allocates a new cluster to replace the one containing the bad sector. If FtDisk is present, NTFS copies the data that FtDisk has recovered into the new cluster to reestablish data redundancy.

Figure 9-37 shows an MFT record for a user file with a bad cluster in one of its data runs. When it receives a bad-sector error, NTFS reassigns the cluster containing the sector to its bad-cluster file. This prevents the bad cluster from being allocated to another file. NTFS then allocates a new cluster for the file and changes the file's VCN-to-LCN mappings to point to the new cluster. This bad-cluster remapping (introduced earlier in this chapter) is illustrated in Figure 9-38 on page 448. Cluster number 1357, which contains the bad sector, is replaced by a new cluster, number 1049 (as you'll see in Figure 9-38).

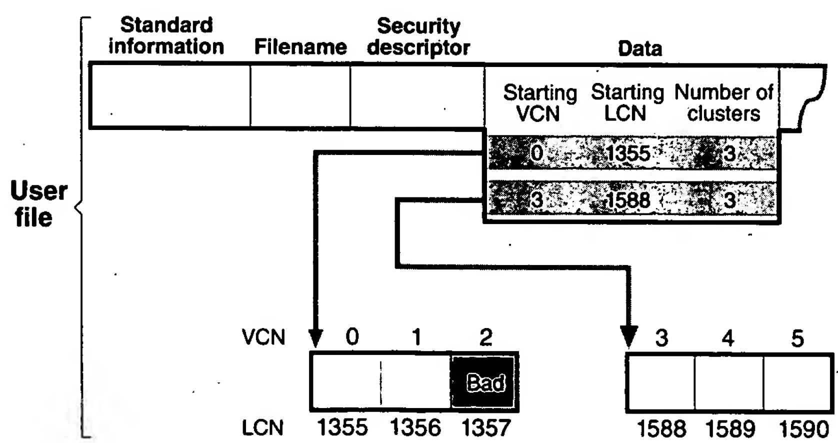


**Figure 9-37**
*MFT record for a user file with a bad cluster*

Bad-sector errors are undesirable, but when they do occur, the combination of NTFS and FtDisk provides the best possible solution. If the bad sector is on a redundant volume, FtDisk recovers the data and replaces the sector if it can. If it can't replace the sector, it returns a warning to NTFS and NTFS replaces the cluster containing the bad sector.

If FtDisk isn't loaded or if the volume isn't configured as a redundant volume, the data in the bad sector can't be recovered. When the volume is formatted as a FAT volume and FtDisk can't recover the data, reading from the bad sector yields indeterminate results. If some of the file system's control structures reside in the bad sector, an entire file or group of files (or potentially, the whole disk) can be lost. At best, some data in the affected file (often, all the data in the file beyond the bad sector) is lost. Moreover, the FAT file system is likely to reallocate the bad sector to the same or another file on the volume, causing the problem to resurface.

Like the other file systems, NTFS can't recover data from a bad sector without help from FtDisk. However, NTFS greatly contains the damage a bad sector can cause. If NTFS discovers the bad sector during a read operation, it remaps the cluster the sector is in, as shown in Figure 9-38 on the following page. If the volume is not configured as a redundant volume, NTFS returns a "data read" error to the calling program. Although the data that was in that cluster is lost, the rest of the file—and the file system—remains intact; the calling program can respond appropriately to the data loss; and the bad cluster won't be reused in future allocations. If NTFS discovers the bad cluster on a write operation rather than a read, NTFS remaps the cluster before writing and thus loses no data and generates no error.

The same recovery procedures are followed if file system data is stored in a sector that goes bad. If the bad sector is on a redundant volume, NTFS replaces the cluster dynamically, using the data recovered by FtDisk. If the volume isn't redundant, the data can't be recovered and NTFS sets a bit in the volume file that indicates corruption on the volume. The NTFS Chkdsk utility checks this bit when the system is next rebooted, and if the bit is set, Chkdsk executes, fixing the file system corruption by reconstructing the NTFS metadata.

In rare instances, file system corruption can occur even on a fault tolerant disk configuration. A double error can destroy both file system data and the means to reconstruct it. If the system crashes while NTFS is writing the mirror copy of an MFT file record, of a filename index, or of the log file, for example, the mirror copy of such file system data might not be fully updated. If the system were rebooted and a bad-sector error occurred on the primary disk at exactly the same location as the incomplete write on the disk mirror,

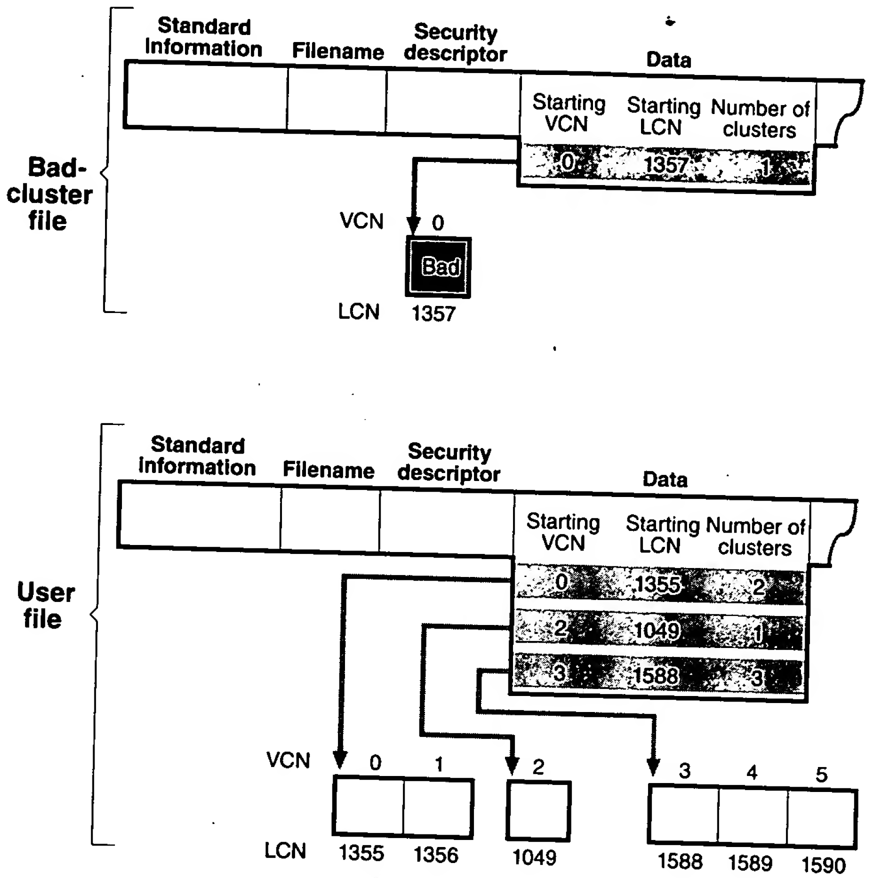


**Figure 9-38**
*Bad-cluster remapping*

NTFS would be unable to recover the correct data from the disk mirror. NTFS implements a special scheme for detecting such corruptions in file system data. If it ever finds an inconsistency, it sets the corruption bit in the volume file, which causes Chkdsk to reconstruct the NTFS metadata when the system is next rebooted. Because file system corruption is rare on a fault tolerant disk configuration, Chkdsk is seldom needed. It is supplied as a safety precaution rather than as a first-line data recovery strategy.

Use of Chkdsk on NTFS is vastly different from its use on the FAT file system. Before writing anything to disk, FAT sets the volume's "dirty bit" and then resets the bit after the modification is complete. If any I/O operation is in progress when the system crashes, the dirty bit is left set and Chkdsk runs when the system is rebooted. On NTFS, Chkdsk runs only when unexpected

or unreadable file system data is found and NTFS can't recover the data from a redundant volume or from redundant file system structures on a single volume. (The system boot sector is duplicated, as are the parts of the MFT required for booting the system and running the NTFS recovery procedure. This redundancy ensures that NTFS will always be able to boot and recover itself.)

Table 9-3 summarizes what happens when a sector goes bad on a disk volume formatted for one of the Windows NT–supported file systems according to various conditions that have been described in this section.

**Table 9-3 Summary of FtDisk and NTFS Data Recovery Scenarios**

| | *FtDisk Installed...* | | *FtDisk Not Installed...* |
|---|---|---|---|
| Scenario | With a SCSI disk that has spare sectors | With a non-SCSI disk or a disk with no spare sectors* | With any kind of disk |
| Fault tolerant volume** | 1. FtDisk recovers the data.<br>2. FtDisk performs *sector sparing* (replaces the bad sector).<br>3. File system remains unaware of the error. | 1. FtDisk recovers the data.<br>2. FtDisk sends the data and a bad-sector error to the file system.<br>3. NTFS performs *cluster remapping*. | N/A |
| Non-fault-tolerant volume | 1. FtDisk can't recover the data.<br>2. FtDisk sends a bad-sector error to the file system.<br>3. NTFS performs *cluster remapping*. Data is lost†. | 1. FtDisk can't recover the data.<br>2. FtDisk sends a bad-sector error to the file system.<br>3. NTFS performs *cluster remapping*. Data is lost†. | 1. Disk driver returns a error to the file system.<br>2. NTFS performs *cluster remapping*. Data is lost†. |

* In neither of these cases can FtDisk perform sector sparing: (1) hard disks that don't use the SCSI protocol have no standard interface for providing sector sparing; (2) some hard disks don't provide hardware support for sector sparing, and SCSI hard disks that do provide sector sparing can eventually run out of spare sectors if a lot of sectors go bad.

** A fault tolerant volume is one of the following: a mirror set or a stripe set with parity.

† In a write operation, no data is lost: NTFS remaps the cluster before the write.

Note that if FtDisk is installed, if the volume on which the bad sector appears is a fault tolerant volume, and if the hard disk is one that supports sector sparing (and that hasn't run out of spare sectors), which file system you are using—FAT or NTFS—doesn't matter. FtDisk replaces the bad sector without the need for user or file system intervention.

If FtDisk is not installed or is installed on a hard disk that doesn't support sector sparing, the file system is responsible for replacing (remapping) the bad sector or—in the case of NTFS—the cluster in which the bad sector resides. The FAT file system does not provide sector or cluster remapping. The benefits of NTFS cluster remapping are that bad spots in a file can be fixed without harm to the file (or harm to the file system, as the case may be) and that the bad cluster won't be reallocated to the same or another file.

## Conclusion

As you saw in the introduction to this chapter, the overriding goal for NTFS was to provide a file system that was not only reliable but also fast. The performance of Windows NT disk I/O is not due solely to the implementation of NTFS, however. It comes in large measure from synergy between NTFS and the Windows NT cache manager. Together, NTFS and the cache manager achieve respectable I/O performance while providing an unprecedented level of reliability and high-end data storage features for both workstation and server systems.